

# Shader Programming with DirectX 9

Heesung Oh

# -차례-

1 Introduction to Shader Programming	-----	1
2 Vertex Shader	-----	5
2.1 Simple Vertex Shader	-----	5
2.2 저 수준 정점 쉐이더 작성 방법	-----	7
2.3 정점 쉐이더 객체와 정점 선언자	-----	10
2.4 변환(Transform)	-----	15
2.5 Vertex Shader Virtual Machine	-----	18
2.6 Texture Mapping	-----	21
2.7 Lighting	-----	24
2.7.1 Lambert 확산	-----	24
2.7.2 Phong 반사	-----	26
2.7.3 Blinn-Phong 반사	-----	30
2.7.4 Lighting Assemble	-----	33
2.8 Vertex Blending	-----	35
2.9 Fog Effect	-----	39
2.10 Toon Shading	-----	43
2.11 윤곽선(Edge)	-----	48
2.12 Depth Encoding	-----	51
2.13 Vertex Shader Effect	-----	52
3 Pixel Shader	-----	59
3.1 Simple Pixel Shader	-----	59
3.1.1 Diffuse 출력	-----	59
3.1.2 상수 레지스터 설정	-----	62
3.1.3 색상 반전(Invert)	-----	63
3.1.4 색상 단색(Monotone)	-----	64
3.2 Texturing & Multi-Texturing	-----	66
3.3 단색 화면(Monotone Effect)	-----	72
3.4 Blur 효과	-----	76
3.5 Pixel Shader Effect	-----	78
3.6 Post Effect Example	-----	80
4 High Level Shading Language 기초	-----	84

4.1 Simple HLSL	-----	85
4.2 HLSL 자료 형	-----	87
4.3 기억 장소(Storage Class)	-----	93
4.4 Semantic	-----	96
4.5 사용자 정의 함수	-----	100
4.6 HLSL 내장 함수, 기타 문법	-----	101
4.7 HLSL for Vertex Processing	-----	102
4.7.1 정점 변환, 텍스처	-----	102
4.7.2 Diffuse color, Fog	-----	105
4.7.3 Lambert Lighting	-----	108
4.7.4 스페큘러 조명	-----	110
4.8 HLSL for Pixel Processing	-----	112
4.8.1 Simple HLSL for Pixel Shader	-----	112
4.8.2 픽셀 처리 사용자 함수	-----	114
4.8.3 텍스처 처리	-----	116
4.8.4 Procedural Texture	-----	123
4.9 ILcEffect 만들기	-----	126
 5 HLSL - ID3DXEffect 응용	-----	129
5.1 HLSL과 ID3DXEffect	-----	129
5.2 Multi-Techniques	-----	132
5.2.1 Effect의 상태 설정과 저 수준 쉐이더	-----	132
5.2.2 Multi-pass	-----	137
5.3 2D Sprite	-----	143
5.4 Effect와 조명	-----	148
5.4.1 분산 조명	-----	149
5.4.2 Specular 효과	-----	153
5.5 NPR(Non-photorealistic rendering) for Lighting	-----	160
5.5.1 Cartoon Shading	-----	160
5.5.2 Hatching	-----	165
5.6 Diffuse and Lighting Mapping	-----	169
5.7 Bump Mapping (Normal Mapping)	-----	173
5.8 Specular Mapping	-----	187
5.9 Environment Mapping	-----	193
5.10 Water Reflection Effect	-----	208
5.11 깊이 버퍼 그림자	-----	211

6 HLSL - Post Effect	-----	220
6.1 모자이크(Mosaic)	-----	221
6.1.1 직사각형 모자이크	-----	221
6.1.2 마름모형 모자이크	-----	224
6.1.3 은행잎 모자이크 1	-----	228
6.1.4 은행잎 모자이크 2	-----	231
6.1.5 직소(Jigsaw) 퍼즐	-----	236
6.1.6 Voronoi Diagram	-----	238
6.2 화면 섭동 (Perturbation)	-----	241
6.2.1 무늬 유리 효과(Embossed Glass) 1	-----	241
6.2.2 무늬 유리 효과 2	-----	245
6.2.3 화면 잡음(Noise)	-----	247
6.3 연필 선 효과(Pencil Stroke)	-----	250
6.4 Distortion (Pencil Stroke)	-----	256
6.4.1 화염 효과(Flame Effect)	-----	256
6.4.2 굴절 효과(Refraction Effect)	-----	272
6.5 흐림 효과	-----	276
6.5.1 흐림 효과(Blur Effect) 개요	-----	276
6.5.2 Glare Effect	-----	286
6.5.3 Cross Filter Effect	-----	288
6.6 수목화 효과	-----	298
6.6.1 외곽선 추출	-----	298
6.6.2 수목화 렌더링	-----	305

# 1 Introduction to Shader Programming

Shader(쉐이더) 프로그래밍은 GPU(Graphic Processing Unit)에 대한 3D 장면 처리를 고정 기능 파이프라인을 이용하지 않고 Assembly 언어와 같은 저 수준 언어 또는 C언어와 같은 고수준 언어(HLSL: High Level Shading Language)로 작성된 프로그램이라 할 수 있습니다.

비디오 제어기(디스플레이 제어기)는 CPU에서 처리한 픽셀 데이터를 화면에 출력하는 단순한 역할에서 메모리의 가격이 낮아지고 컴퓨터의 하드웨어 성능이 발전하면서 화면에 출력할 픽셀의 데이터 일부를 CPU 대신 처리하게 되어 단순 기능의 비디오 제어기 대신 디스플레이 프로세서(Display Processor)로 발전하게 되었습니다. 이 디스플레이 프로세서가 라이팅, 정점 변환, 픽셀 샘플링 등을 CPU 대신 처리하게 되었는데 이것을 CPU와 대응되는 개념으로 그래픽 출력을 전달하는 장치를 GPU(Graphic Processing Unit)라고 합니다.

처음에 GPU는 기술적인 한계와 비용에 의해서 그래픽에 대한 모든 처리 과정이 고정(Fixed) 되어 있었습니다. 이것을 고정 기능 파이프라인(Fixed Function Pipeline)으로 부릅니다. GPU도 발전하면서 고정 기능 파이프라인의 일부를 사용자가 프로그램을 작성할 수 있도록 구성되게 되었는데 이것을 프로그램 가능한 파이프라인(Programmable Pipeline)으로 부르게 되었으며 쉐이더는 GPU가 제공하는 Programmable Pipeline의 정점과 Pixel의 처리를 사용자의 프로그램으로 처리하는 것입니다.

C/C++ 같은 고급 언어는 inline Assembly를 이용해서 저급 언어도 일부 지원하고 있습니다.

```
#include <stdio.h>
...
char* s1 = "Hello";
char* s2 = "World";
char* f = "%s %s\n";
_asm {
    mov eax, s2;
    push eax;
    mov ebx, s1;
    push ebx;
    mov ecx, f;
    push ecx;
    call printf;
```

만약 여러분이 Microsoft의 DirectX의 Shader를 사용하면 C언어와 유사한 문법으로 쉐이더 프로그

램을 쉽게 작성할 수 있습니다. 또한 DirectX의 쉐이더는 고급 언어뿐만 아니라 어셈블리어 형태의 저급 언어 문법이 지원이 되어 동시에 활용할 수도 있습니다.

우리가 C/C++ 등으로 작성하는 프로그램은 실제로 CPU가 처리하는 명령어들입니다. 그런데 우리는 CPU를 완벽하게 이해하지 않고도 프로그램을 작성할 수 있습니다. 이것은 컴파일러라는 도구에 의해서 사람과 가까운 형태의 언어를 CPU가 처리할 수 있는 기계 코드로 바꾸어 주기 때문입니다. GPU도 제조하는 회사마다 고유의 명령어들이 있을 수 있어서 같은 계열이 아니면 다른 문법 또는 다른 명령어로 프로그램을 작성해야 합니다. 그런데 DirectX Shader를 지원하는 그래픽 카드에 대해서는 여러분이 쉐이더 프로그램을 작성할 때 그래픽 카드의 특성과 세부적인 내용을 생각하지 않고 동일한 문법으로 GPU를 프로그램 할 수 있습니다.

고정 기능 파이프라인을 가지고 게임 프로그램을 만들 때 발생하는 문제들은 단순히 값을 바꾸어서 해결되는 경우도 있습니다. 하지만 쉐이더는 이렇게 단순히 값을 바꾸는 반복 작업으로 문제 해결이 잘 안됩니다. 이것은 쉐이더는 GPU의 처리 과정을 직접 작성하는 것이라서 각 단계의 처리 절차를 알고 있지 않으면 문제의 원인과 해결점을 찾을 수 없기 때문입니다.

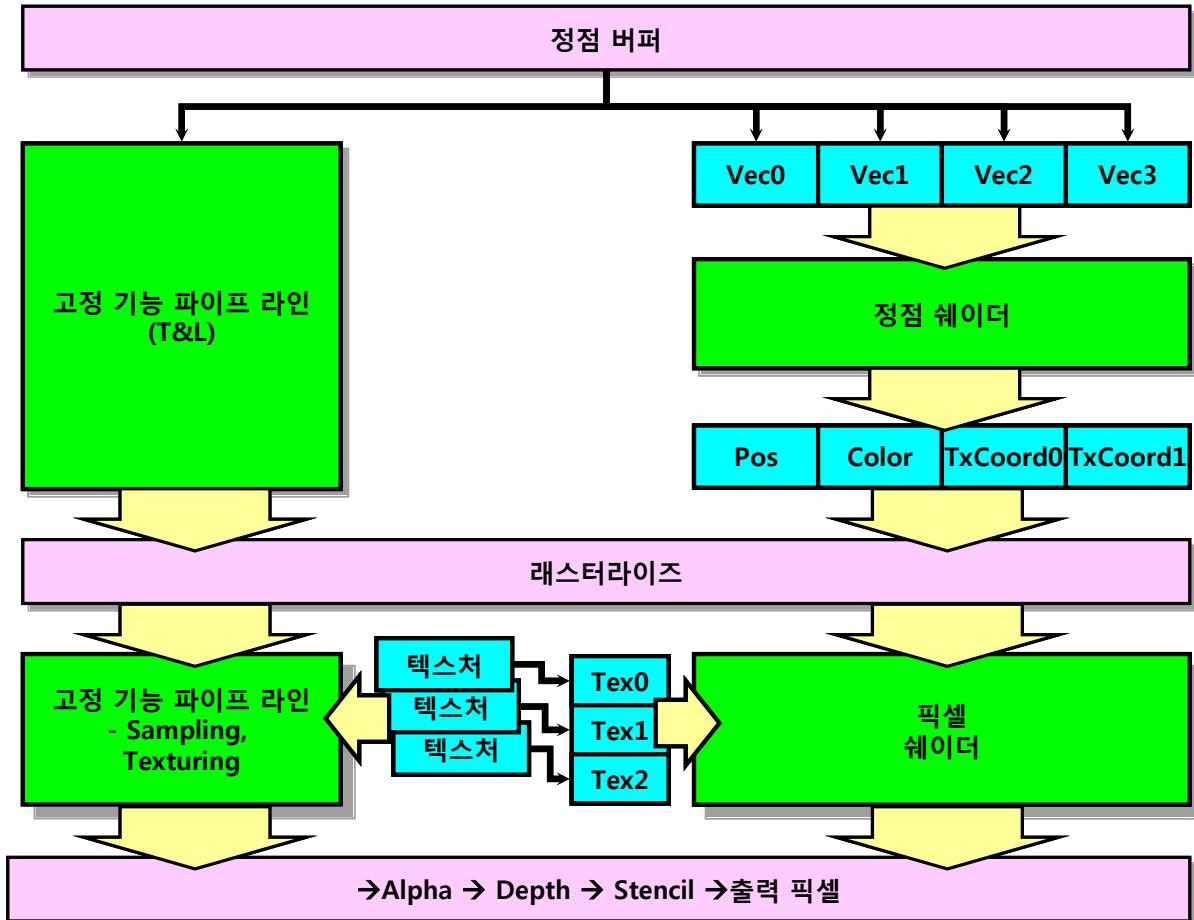
여러분이 능숙하게 쉐이더를 사용할 수 있는 능력을 키우는 가장 확실한 방법은 먼저 GPU의 처리 과정을 분명하게 이해 하는 것입니다. GPU의 처리 과정만 이해해도 여러분은 올바른 쉐이더 프로그램을 작성할 수 있을 뿐만 아니라 다른 사람이 작성한 것도 쉽게 활용할 수 있습니다.

또한 선형 대수(Linear Algebra) 정도의 수학적 지식이 있어야 합니다. DirectX Shader에서 제공되는 함수들은 거의 기본적인 함수들 밖에 없습니다. 여러분은 이 기본적으로 제공되는 함수들을 조합해서 정점의 변환, 조명 효과, 퍽셀 처리 등을 작성해야 합니다. 예를 들어 조명 효과의 풍반사는 쉐이더로 작성한다면 불과 몇 줄도 안되지만 여러분이 벡터의 내적과 power() 함수를 알지 못하면 그림의 떡일 수 밖에 없습니다. 따라서 수학적인 능력이 부족하다 느끼면 틈틈이 게임에 관련된 수학을 찾아보고 연습하기 바랍니다.

GPU의 처리 과정을 이해하는 여러 방법 중에서 고정 기능 파이프라인에서 작성한 것을 쉐이더로 프로그램 가능한 파이프라인에 맞게 바꾸어 보는 것이 가장 쉽고 빠릅니다. 고정 기능 파이프라인의 처리 과정을 간단히 복습하면 3D 장면을 연출하기 위해 그래픽 파이프라인에 입력된 정점 데이터는 최초로 정점 처리(Vertex Processing)을 진행 합니다. 정점 처리 과정은 정점의 월드변환 → 정점 블렌딩 → 카메라 공간으로 뷰 변환 → 포그 결합 → 조명 & 재질 결합 → 정규 변환(장치 독립의 정규좌표 변환) → 뷰 포트(View port) 변환 → 래스터라이징(Rasterizing) 변환입니다.

래스터라이징을 거치면 정점의 데이터는 퍽셀로 바뀌게 됩니다. 이러한 데이터는 또다시 퍽셀 처리(Pixel Processing)을 거치며 퍽셀 처리는 샘플링(Sampling) → Multi texturing → Alpha Test → Depth Test → Stencil Test → Pixel fog → Alpha Blending 등을 거쳐 최종적으로 백 버퍼에

저장이 됩니다.



<고정 기능 파이프라인과 프로그램 가능한 파이프라인 비교>

이 중에서 정점의 월드변환에서 레스터라이징 전까지에 해당하는 과정을 프로그램 하는 것을 정점 쉐이더(Vertex Shader) 프로그래밍이라 합니다. 그리고 샘플링에서 멀티 텍스처링까지 픽셀 처리에 대한 프로그램을 픽셀 쉐이더(Pixel Shader) 프로그래밍이라 합니다. 즉, 정점 쉐이더와 픽셀 쉐이더는 정점 처리와 픽셀 처리의 일부만을 프로그램 하는 것을 의미합니다.

또한 정점 쉐이더로 처리하는 과정과 픽셀 쉐이더로 처리하는 과정은 독립적으로 진행을 합니다. 이것은 또한 고정 기능 파이프라인의 픽셀 처리와 프로그램 가능한 파이프라인의 픽셀 처리에서 입력되는 정점 처리 후의 데이터가 정점 쉐이더로 처리되었던 결과인지 아니면 고정 기능 파이프라인으로 처리되었던 결과인지를 구분하지 않는다는 것입니다. 이러한 이유로 때로는 정점 처리는 고정 기능 파이프라인으로 처리하고 픽셀 처리는 픽셀 쉐이더를 사용하거나 정점 쉐이더를 사용하고 픽셀 처리는 고정 기능 파이프라인으로 처리하기도 합니다.

같은 개수의 명령어를 처리하는 것이라면 고정 기능 파이프라인이 더 빠를 수 있다고 할 수 있지

만 쉐이더를 사용하더라도 "느려지지 않는다."라는 것입니다. 우리가 쉐이더를 사용하는 가장 큰 이유는 고정 기능 파이프라인에서 처리하지 못하는 내용을 쉐이더 프로그램을 통해 극복하는 것입니다.

지금까지 대충 쉐이더의 역할과 내용을 살펴보았습니다. 이제 본격적으로 정점 쉐이더와 퍽셀 쉐이더를 공부해봅시다.

## 2 Vertex Shader

### 2.1 Simple Vertex Shader

정점 쉐이더 프로그래밍은 정점 처리 과정을 프로그램 가능한 파이프라인에 대한 프로그램입니다. 쉐이더가 처음 만들어졌을 때는 GPU에서 지원되는 기능이 많지 않아서 Assembly 형태의 언어로 작성해야만 했습니다.

다음 코드는 C언어의 "Hello world"에 해당하는 가장 간단한 정점 쉐이더 코드입니다.

```
"vs_1_1           Wn"
"dcl_position    v0      Wn"
"mov oPos, v0    Wn"
```

쉐이더를 어셈블리 형태의 저급 언어로 작성할 때 제일 먼저 여러분은 쉐이더 컴파일 버전을 명시해야 합니다. 첫 번째 줄의 vs\_1\_1(또는 vs.1.1)은 정점 쉐이더 버전 1.1로 쉐이더를 작성했다는 의미이며 이 버전을 이용해서 이후의 쉐이더 코드를 컴파일 합니다.

정점 쉐이더 버전은 1.1, 1.2, 1.3, 2.0, 2.x 3.0 등이 있으며 쉐이더 명령어는 버전마다 명령어들이 조금씩 다르기 때문에 이것을 어떤 쉐이더 버전을 사용하고 있는지 꼭 명시해야 합니다.

두 번째 줄의 "dcl\_"으로 시작되는 문장은 GPU에 전달되는 정점 데이터가 처리되기 전에 임시로 머무는 입력 값을 선언하고 저장하는 레지스터(v로 시작)를 지정하는 것입니다.

앞의 "dcl\_position v0"는 GPU에 입력된 정점 데이터의 위치를 v0 레지스터에 지정한다 의미입니다. 입력 값을 저장하는 레지스터는 v0~v12까지 있습니다.

세 번째 줄의 mov는 데이터의 복사를 지시하는 명령어입니다. "mov oPos, v0"는 v0에 저장된 x, y, z, w 출력 레지스터 oPos에 저장하라는 의미입니다. 여기서 중요한 것은 정점 쉐이더에서 처리한 후의 위치에 대한 결과 값은 반드시 출력 레지스터에 복사를 해야 한다는 것입니다. 만약 여러분이 출력 레지스터에 어떤 값도 지정하지 않으면 쉐이더 프로그램은 컴파일이 완료되지 않습니다.



<간단한 정점 쉐이더. [s0v\\_01\\_vertex01.zip](#)>

앞의 쉐이더 코드는 입력레지스터에 저장된 정점 위치를 그대로 출력 레지스터에 저장하도록 했습니다. 만약 여러분이 정점의 좌표를  $[-1, -1, 0] \sim [1, 1, 1]$  범위로 작성했다면 앞의 그림과 같은 삼각형을 출력 할 수 있습니다. 대부분의 그래픽 카드는 색상을 지정하지 않으면 흰색을 출력하는데 간혹 앞의 오른쪽 그림과 같이 검정색 삼각형이 출력 될 수도 있습니다.

[s0v\\_01\\_vertex01.zip](#)의 CShaderEx::Create() 함수에서 삼각형 출력에 대한 정점 쉐이더 작성 예를 볼 수 있습니다.

이번에는 이전의 코드에 색상이 출력되도록 쉐이더 코드를 작성해 봅시다. 고정기능 파이프라인에서는 정점에 색상이 있어야 되지만 쉐이더는 직접 색상을 지정할 수 있습니다. 이렇게 직접 쉐이더 색상을 지정하려면 상수 레지스터를 이용해야 합니다. 상수 레지스터의 값을 설정하는 방법은 쉐이더 코드에서 지정하는 방법과 외부에서 지정하는 방법 2가지가 있습니다. 다음은 쉐이더 코드 내부에서 상수 레지스터에 색상을 지정하고 출력하는 예입니다.

```
"vs_1_1           Wn"
"def c10, 1, 1, 0, 1   Wn"
"dcl_position    v0   Wn"
"mov oPos, v0      Wn"
"mov oD0, c10      Wn"
```

두 번째 줄의 def는 상수 레지스터에 값을 지정하는 명령어로 "def c10, 1, 1, 0, 1"은 상수 레지스터 c10에 r, g, b, a = (1, 1, 0, 1)로 저장하라는 명령어입니다.

상수 레지스터는 c로 시작을 하며 c는 constant의 첫 글자입니다. 마지막의 "mov oD0, c10"은 출력 레지스터 oD0에 상수레지스터에 저장된 값을 복사를 지시하는 것입니다.

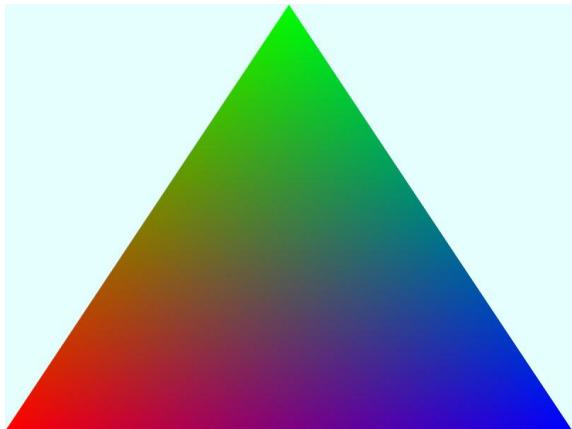


<상수 레지스터를 사용한 색상 출력. [s0v\\_01\\_vertex02.zip](#)>

이 번에는 정점이 색상을 가지고 있다고 가정하고 정점의 색상을 출력해 봅시다. 여러분은 이전 쉐이더 코드에서 상수 레지스터의 값 대신 입력 레지스터에 정점 색상을 지정하고 이것을 출력 레지스터 oD0에 복사하면 됩니다.

```
"vs_1_1           \n"
"dcl_position    v0 \n"
"dcl_color       v1 \n"
"mov oPos, v0    \n"
"mov oD0, v1    \n"
```

세 번째 줄의 "dcl\_color v1" 정점의 Diffuse 값을 입력 레지스터 v0에 저장을 지시하는 것이며 마지막 "mov oD0, v1"은 출력 레지스터 oD0에 v1의 값 복사를 지시하는 것입니다.



<정점의 Diffuse 출력. [s0v\\_01\\_vertex03.zip](#)>

## 2.2 저 수준 정점 쉐이더 작성 방법

어셈블리어 형태의 저 수준 쉐이더 문법은 간단하게 구성되어 있어서 일정한 순서와 명령어들을 익히면 쉽게 쉐이더 코드를 작성할 수 있습니다. 저 수준 쉐이더 코드 작성은 다음과 같이 총 5단계로 나누어서 작성됩니다.

1. 정점 쉐이더 버전을 선언한다.
2. 필요하다면 상수 레지스터의 값을 지정한다.
3. 입력 레지스터를 선언하고 지정한다.
4. 정점의 위치, 법선 벡터, 색상, 안개 효과 값들에 대한 연산을 한다.
5. 연산한 결과를 출력 레지스터에 저장한다.

첫 번째 단계의 쉐이더 버전 선언은 "vs"로 시작을 하며 vs\_1\_1(또는 vs.1.1), vs\_1\_2(또는 vs.1.2), vs\_1\_3, vs\_2\_0, vs\_3\_0 등으로 작성 합니다.

2 번째 단계인 상수 레지스터에 값을 설정하는 것은 "def"로 시작합니다. 쉐이더는 float 형 4개를 하나의 데이터의 단위로 처리합니다. 따라서 상수 레지스터에 값을 지정할 때는 float형 값 4개를 지정해야 합니다.

```
def c0, 1.0, 3.0, 0.0, 2 ; 상수 레지스터 c0에 (1, 3, 0, 2) 값 저장
def c12, 1.0, 0.0, 0.0, 1 ; 상수 레지스터 c12에 (1, 0, 0, 1) 값 저장
```

상수 레지스터는 c로 시작을 하며 상수레지스터의 개수는 쉐이더 버전마다 다르며 상수 레지스터에 저장된 값은 쉐이더 코드 처리 과정에서 수정할 수 없는 읽기 전용입니다.

저 수준 쉐이더 문법에서 주석은 Assembly 언어의 주석 ";"과 C 언어의 주석("//", "/\* \*/" 이 있습니다.

3 번째 단계는 입력 레지스터를 선언하고 지정하는 것으로 "dcl\_"로 시작합니다. dcl은 declare를 의미합니다. 가장 많이 사용되는 정점의 위치, 법선 벡터, Diffuse, 텍스처 좌표를 지정할 때는 다음과 같이 작성합니다.

```
dcl_position v0      ; 입력 정점의 위치를 v0 레지스터로 선언(저장)
dcl_normal    v1      ; 입력 정점 법선을 v1 레지스터로 선언(저장)
dcl_color     v2      ; 정점의 diffuse 색상을 v2 레지스터로 선언(저장)
dcl_texcoord v3      ; 정점의 텍스처 좌표를 v3 레지스터로 선언(저장)
```

이렇게 "dcl\_" 로 입력 레지스터를 지정하면 정점 데이터는 위치(position), 법선(normal), Diffuse(color, color0), 텍스처 좌표(texcoord, texcoord0~7) 등으로 분리되어 "dcl\_"에서 지정된 레지스터에 저장 됩니다. "dcl\_..." 끝에는 숫자가 붙을 수 있습니다. 숫자 0은 숫자가 없을 때와 동등합니다. 즉, dcl\_color0과 dcl\_color는 같은 내용이며 Diffuse를 지정할 때 사용합니다.

정점의 스페큘러(Specular)는 dcl\_color1로 합니다. 텍스처 좌표도 여러 개를 가지면 각각 dcl\_texcoord0~7를 사용하면 됩니다. 때로는 정점을 구성하는 위치 또는 법선 벡터를 여러 개 둘 수 있습니다. 마찬가지 방법으로 dcl\_position[#], dcl\_normal[#]으로 숫자를 붙여서 사용합니다.

"dcl\_" 다음에 붙는 키워드는 D3DDECLUSAGE 열거 형에서 "D3DDECLUSAGE\_" 제외한 나머지를 사용하면 됩니다.

4 번째 단계에서는 입력 레지스터 값과 상수 레지스터 값을 이용해서 연산을 수행합니다. 연산의

문법은 Assembly와 거의 같으며 명령어는 Mnemonic 형태입니다.

`operator dest Register, src1 Register, src2 Register, src3 Register`

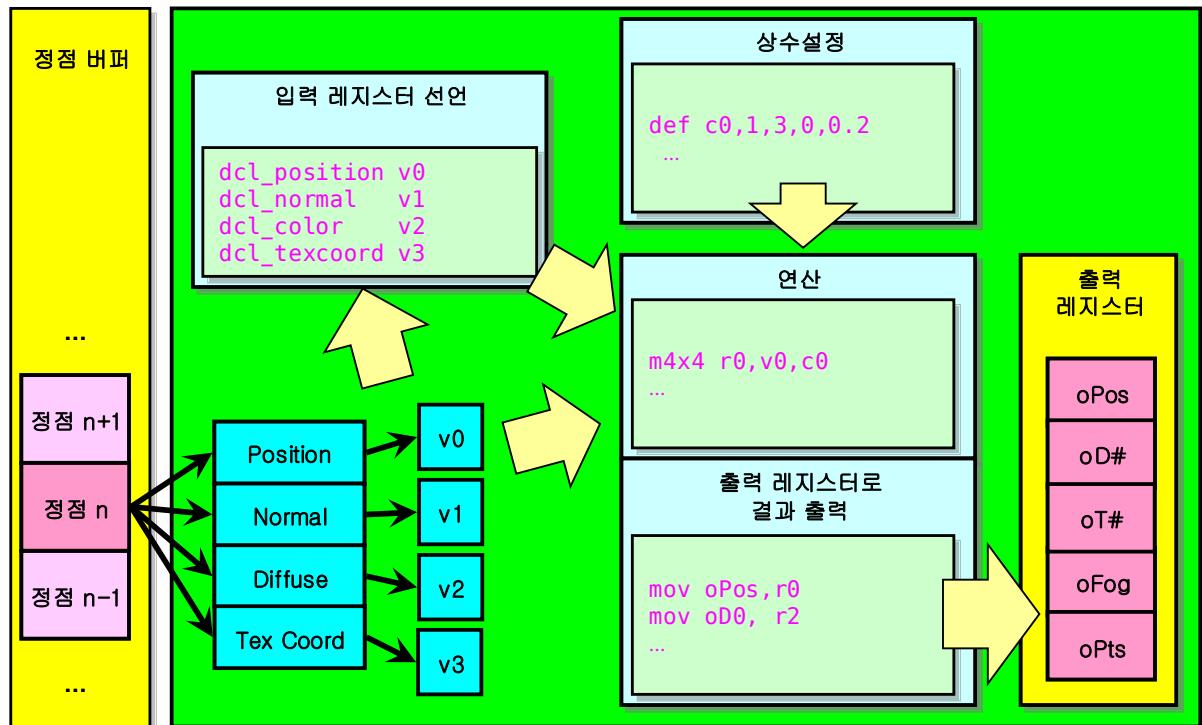
dest는 임시(r#), 출력(o...) 레지스터만 가능하며 상수(c#), 입력(v#)는 읽기 전용이기 때문에 허용하지 않습니다. operator의 모든 연산은 float4를 x, y, z, w 또는 r, g, b, a로 나누어서 연산이 가능합니다. 또한 xyzw 순서를 바꾸거나 yyxx와 같이 성분의 순서와 요소를 섞을 수 있으며 이것을 swizzling이라 합니다. swizzling은 source에서만 가능합니다.

연산을 하면서 결과를 임시로 저장할 수 있습니다. 이 경우에 임시 레지스터(Temporary Register)를 사용할 수 있습니다. 임시 레지스터는 "r"로 시작을 합니다. 다음은 swizzling으로 임시 레지스터 "r0"에 입력 레지스터 "v1"의 값이 복사하는 예입니다.

`mov r0, v1.zy`

이렇게 하면 r0(x, y, z, w)는 (v1.z, v1.y, v1.y, v1.y) 값이 저장 됩니다.

5 번째 단계에서는 출력 레지스터로 결과를 저장합니다. 출력 레지스터는 소문자 "o"로 시작을 하며 여러분은 출력 레지스터 oPos에 값을 반드시 저장해야 합니다.



<정점 쉐이더 레이아웃>

Diffuse 값은 oD0, 스페큘러 값은 oD1에 저장을 하고, 텍스처 좌표는 oT0~ oT7에 저장합니다. 안개 값은 oFog, Point의 크기는 oPts에 저장합니다.

```
mov oPos, r0
mov oDo , v2
mov oT0 , v3
...
...
```

이렇게 사용자가 작성한 쉐이더 코드는 다음 그림과 같이 프로그램 가능한 파이프라인에서 정점 데이터가 입력이 되면 쉐이더 코드의 명령어 순서에 따라 각각의 동작을 수행 합니다.

지금까지 간단한 정점 쉐이더와 쉐이더 작성 방법을 살펴 보았습니다. 다음으로 정점 쉐이더를 사용해서 정점의 처리 과정을 살펴봅시다.

## 2.3 정점 쉐이더 객체와 정점 선언자

프로그램 가능한 파이프라인에 우리가 작성한 정점 쉐이더 코드를 적용하려면 정점 쉐이더(Vertex Shader) 객체와 정점 선언(Vertex Declaration) 객체 2개가 필요합니다.

정점 쉐이더 객체는 컴파일 한 쉐이더 코드를 적재한 객체이고, 정점 선언 객체는 프로그램 가능한 파이프라인에 정점 데이터의 형식을 알리는 객체입니다. 이것은 고정기능 파이프라인의 SetFVF() 함수를 대신한다고 할 수 있습니다.

정점 쉐이더 객체를 생성하기 위해서 먼저 정점 쉐이더 코드를 컴파일 해야 합니다. D3DXAssembleShader() 함수는 저 수준 언어로 작성된 쉐이더 코드를 컴파일 하며, 컴파일 과정에서 발생한 문법 오류 등을 검사해 줍니다.

```
DWORD dwFlags = 0;
#if defined( _DEBUG ) || defined( DEBUG )
    dwFlags |= D3DXSHADER_DEBUG;
#endif
```

```
LPD3DXBUFFER pShd     = NULL;
LPD3DXBUFFER pErr     = NULL;
INT          iLen      = strlen(sShader);
```

```
hr = D3DXAssembleShader(sShader, iLen, NULL, NULL, dwFlags, &pShd, &pErr);
```

만약 파일에서 작성한 저 수준 쉐이더를 컴파일 하려면 D3DXAssembleShaderFromFile() 함수를 다음과 같이 사용합니다.

```
hr = D3DXAssembleShaderFromFile("파일 이름", NULL, NULL, dwFlags, &pShd, &pErr);
```

앞의 예처럼 D3DXAssembleShader() 함수의 마지막에 오류 정보를 저장할 ID3DXBuffer 형 인수를 넣어주면 컴파일 과정에서 발생한 에러 내용과 위치(행)을 문자열로 저장해 줍니다. 우리는 이것을 다음의 (char\*)pErr->GetBufferPointer()와 같이 문자열로 캐스팅에서 문제점을 파악할 수 있습니다.

```
if ( FAILED(hr) )
{
    if(pErr)
    {
        char* sErr = (char*)pErr->GetBufferPointer();
        MessageBox( hWnd, sErr, "Err", MB_ICONWARNING );
        pErr->Release();
    }
    return -1;
}
```

D3DXAssembleShader() 함수는 컴파일만 담당하기 때문에 컴파일 한 쉐이더 명령어를 사용하기 위해서 결과를 메모리에 적재해야 합니다. 정점 쉐이더 코드가 적재된 객체를 정점 쉐이더 (IDirect3DVertexShader9) 객체라 하는데 이 객체는 다음과 같이 D3D 디바이스의 CreateVertexShader() 함수에 컴파일 한 결과를 인수로 전달하면 생성 됩니다.

```
IDirect3DVertexShader9* m_pVs; // 정점 쉐이더 객체
...
hr = m_pDev->CreateVertexShader( (DWORD*)pShd->GetBufferPointer() , &m_pVs);
if ( FAILED(hr) )
```

정점 데이터와 쉐이더 코드의 관계는 그림처럼 입력 레지스터 선언에 의해서 데이터가 분해되어 입력 레지스터에 저장됩니다. 이 입력 레지스터에 저장된 값과 미리 설정된 상수 레지스터에 저장된 값을 가지고 연산을 합니다. 이 연산의 결과를 임시 레지스터에 저장하거나 직접 출력 레지

스터로 복사되고 출력 레지스터 내용은 Rasterizing 또는 Pixel Processing에서 처리 됩니다.

```
return -1;
```

프로그램 가능한 파이프라인에서는 고정 기능 파이프라인의 FVF(Flexible Vertex Format) 상수를 사용할 수 없습니다. 그 대신 정점 선언(자) 객체를 사용하는데 정점 선언 객체를 생성하기 위해서 먼저 D3DVERTEXELEMENT9 구조체 변수를 MAX\_FVF\_DECL\_SIZE 만큼 배열로 선언하고 manual로 값을 채웁니다.

```
D3DVERTEXELEMENT9 vertex_decl[MAX_FVF_DECL_SIZE] =
{
    { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0 },
    { 0, 12, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR, 0 },
    ...
    D3DDECL_END()
};
```

구조체를 채울 때 마지막은 "D3DDECL\_END()"을 사용해서 더 이상 추가가 없음을 지시합니다. 이렇게 manual로 D3DVERTEXELEMENT9의 값을 설정하는 것도 좋겠지만 처음 하는 분들에게는 부담이 됩니다. DXSDK의 D3DXDeclaratorFromFVF() 함수는 이미 알려진 FVF 값에서 D3DVERTEXELEMENT9 구조체 변수를 채우는 함수로 이 함수를 사용하는 것이 가장 무난합니다.

manual로 작성하는 경우는 고정 기능 파이프라인에서 제공 되지 않는 FVF를 사용할 때, 예를 들어 정점의 위치가 2개 이상 이거나, 여러 개의 법선 벡터를 사용하게 되면 직접 D3DVERTEXELEMENT9 값을 작성해야 하며 쉐이더 코드 작성에서도 dcl\_position 또는 dcl\_normal 끝에 숫자를 붙여서 작성해야 합니다.

```
IDirect3DVertexDeclaration9* m_pFVF;           // 정점 선언 객체
...
D3DVERTEXELEMENT9 vertex_decl[MAX_FVF_DECL_SIZE] ={0};
D3DXDeclaratorFromFVF(VtxD::FVF, vertex_decl);
if(FAILED(m_pDev->CreateVertexDeclaration( vertex_decl, &m_pFVF )))
    return -1;
```

이렇게 프로그램 가능한 파이프라인을 사용하기 위해서 정점 쉐이더 객체와 정점 선언 객체를 만들었습니다. 이들을 이용해서 렌더링 하는 순서는 다음과 같습니다.

1. 정점 쉐이더와 정점 선언 객체 사용 명시함으로써 프로그램 가능한 파이프라인 사용을 알린다.
2. 정점 선언 객체로 정점 데이터의 형식을 파이프라인에 알린다.

3. 필요하면 상수 레지스터 값을 설정한다.
4. 정점을 그린다.
5. 정점 쉐이더 객체 사용을 해제해서 고정 기능 파이프라인 사용으로 돌아온다.

프로그램 가능한 파이프라인을 사용하려면 정점 쉐이더 사용을 다음과 같이 명시해야 합니다.

```
m_pDev->SetVertexShader(m_pVs);
```

이렇게 하면 GPU는 고정 기능 파이프라인 대신에 프로그램 가능한 파이프라인으로 정점 데이터를 전달합니다.

다음으로 정점 선언 객체를 지정해서 입력된 정점 데이터의 형식을 파이프라인에 알립니다.

```
m_pDev->SetVertexDeclaration( m_pFVF );
```

필요한 상수의 전달입니다. CPU에서 데이터의 처리는 int 형 이듯이 쉐이더는 float4 (float \*4) 형 입니다. 파이프라인에 상수를 전달하는 함수는 디바이스의 SetVertexShader() 함수입니다. 이 함수의 사용은 다음과 같습니다.

```
m_pDev->SetVertexShaderConstantF( 0, (FLOAT*)&mVP , 4);
```

SetVertexShaderConstantF() 함수의 F는 FLOAT 형 데이터 전달을 의미합니다. 함수의 첫 번째 인수는 상수 레지스터 이름의 c를 제외한 나머지로 앞의 코드처럼 "0"으로 지정하면 c0 상수 레지스터에 값을 지정하는 것입니다. 함수의 두 번째 인수는 상수에 설정할 값의 주소를 지정합니다. 세 번째 인수는 float4형의 개수를 의미하며 앞의 코드는 float4 형이 4개 전달되므로 총 16개의 float 형 데이터를 상수 레지스터에 지정하는 것입니다. 앞서 쉐이더의 레지스터는 float4형이 기본이라고 했습니다. 그런데 데이터가 16개가 전달되어서 c0는 float 형 4개가 적재되고 나머지 12개의 데이터는 c1, c2, c3 레지스터에 순차적으로 기록이 됩니다.

상수 연결에서 주의할 내용은 float 형 데이터 하나를 설정 하더라도 쉐이더는 float4가 기본이어서 데이터를 float4로 늘린 후에 값을 전달해야 합니다. 예를 들어 상수 레지스터 c10에 정점의 위치 vcP(=float2: x, y)으로 구성된 데이터를 연결한다 하더라도 다음과 같이 코딩 해야 합니다.

```
FLOAT p[4] = {vcP.x, vcP.y, 0, 0};  
m_pDev->SetVertexShaderConstantF( 10, (FLOAT*)&p , 1);
```

저 수준에서 행렬 데이터를 상수 레지스터에 복사할 경우에 행렬의 값을 전치(Transpose) 시켜야 합니다. 이것은 쉐이더에서 행렬의 연산 과정이 마치 오른손 좌표계 연산과 동일해서 왼손 좌표를 사용하는 DirectX의 행렬 값을 전치해야 올바른 결과를 만들 수 있습니다.

```
D3DXMATRIX mtVP = mtWld * mtViw * mtPrj;
D3DXMatrixTranspose( &mtVP , &mtVP );
m_pDev->SetVertexShaderConstantF( 0, (FLOAT*)&mtVP , 4);
```

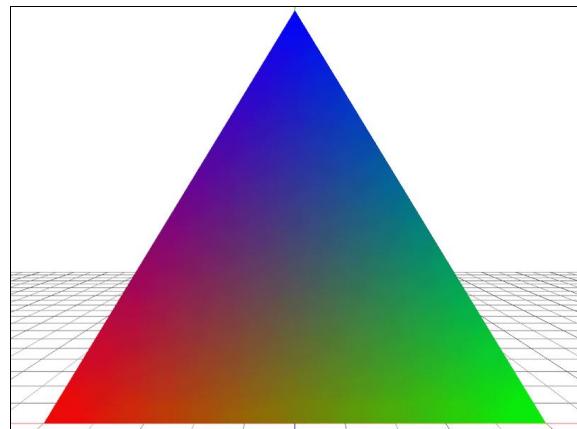
고 수준 언어인 HLSL을 사용하게 되면 행렬을 전치하지 않고 그대로 사용할 수 있습니다.

상수 레지스터는 float형 이외에 정수형과 bool형을 지정할 수 있습니다. 정수형은 SetVertexShaderConstantI() 함수를 사용하고 bool 형은 SetVertexShaderConstantB() 함수를 사용합니다.

상수 레지스터 지정 후에 DrawPrimitive()함수 등을 호출해서 렌더링을 하며, 렌더링을 끝내고 고정 파이프라인으로 돌아오기 위해서 다음과 같이 SetVertexShader()함수에 NULL 을 설정합니다.

```
m_pDev->SetVertexShader( NULL );
```

[s0v\\_02\\_shader\\_string.zip](#)는 문자열로 작성된 쉐이더에 대한 예제이고 [s0v\\_02\\_shader\\_file.zip](#) 은 파일에서 작성한 쉐이더 예제입니다. 이 둘의 CShaderEx 클래스는 프로그램 가능한 파이프라인을 이용해서 월드 공간에서 삼각형을 렌더링 합니다.



<프로그램 가능한 파이프라인에서 삼각형 출력:

[s0v\\_02\\_shader\\_string.zip](#), [s0v\\_02\\_shader\\_file.zip](#)

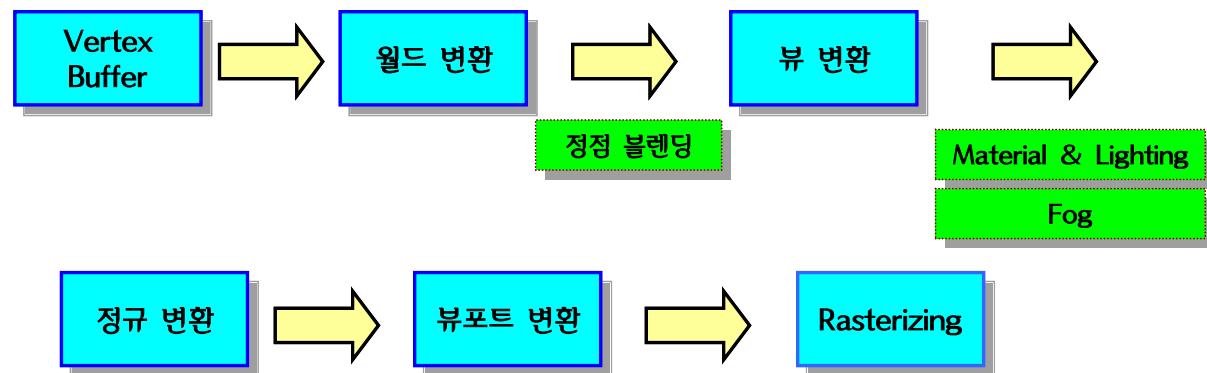
지금까지 정점 쉐이더 객체와 정점 선언 객체를 생성하고 프로그램 가능한 파이프라인을 이용하고 상수 레지스터 설정 방법을 간단히 살펴보았습니다. 정점 쉐이더를 사용과 관련된 함수들을 정리한다면 다음과 같습니다.

1. 파일 또는 문자열로 어셈블리어 형식의 저 수준 쉐이더 명령어들을 작성한다.
2. 저 수준 쉐이더 명령어들을 컴파일 한다. → D3DXAssembleShader() 함수
3. 컴파일 된 쉐이더 명령어로 정점 쉐이더 객체를 생성한다. → pDevice->CreateVertexShader()

4. 정점 형식 객체를 생성한다. → pDevice->CreateVertexDeclaration()
5. GPU에게 프로그램 가능한 파이프라인 사용을 알린다. → pDevice->SetVertexShader()
5. 파이프라인에 입력된 정점 데이터 형식을 알린다. → pDevice->SetVertexDeclaration()
6. 파이프라인의 상수 레지스터를 설정한다. → pDevice->SetVertexShaderConstant{F|B|I|}()
7. 장면을 그린다. → pDevice->DrawPrimitive()
8. 정점 쉐이더 사용을 해제한다. → pDevice->SetVertexShader(NULL)

## 2.4 변환(Transform)

3D 기초 과정으로 돌아서 고정 기능 파이프라인의 정점 처리 과정을 보면 크게 정점의 변환, 조명 효과 적용, 안개 효과 적용입니다.



<쉐이더 프로그램 대상인 고정 기능 파이프라인의 정점 처리 과정>

정점의 변환은 다시 월드 변환, 뷰 변환, 정규 변환으로 나눌 수 있으며 수식으로 다음과 같이 정리할 수 있습니다.

정점의 정규 변환 위치' = 정점의 위치 \* (월드 행렬 \* 뷰 행렬 \* 투영 행렬)

이 수식을 다음과 같이 [s0v\\_02\\_shader\\_string.zip](#), [s0v\\_02\\_shader\\_file.zip](#)에 구현하고 있습니다.

```

vs_1_1
dcl_position    v0
dcl_color      v1
m4x4 oPos, v0, c0
mov  oD0, v1
  
```

4 번째 줄의 m4x4는 벡터와 행렬의 곱셈 연산을 지시하는 명령어로 "m4x4 oPos, v0, c0" 상수 레

지스터 c0, c1, c2, c3를 행으로 구성하는 행렬과 입력 레지스터 v0를 곱해서 oPos에 저장하는 것이며 수식으로 표현하면 다음과 같습니다.

$$\text{m4x4 } \text{oPos, v0, c0} \rightarrow \begin{bmatrix} \text{oPos.x} \\ \text{oPos.y} \\ \text{oPos.z} \\ \text{oPos.w} \end{bmatrix} = \begin{bmatrix} \text{c0.x} & \text{c0.y} & \text{c0.z} & \text{c0.w} \\ \text{c1.x} & \text{c1.y} & \text{c1.z} & \text{c1.w} \\ \text{c2.x} & \text{c2.y} & \text{c2.z} & \text{c2.w} \\ \text{c3.x} & \text{c3.y} & \text{c3.z} & \text{c3.w} \end{bmatrix} \begin{bmatrix} \text{v0.x} \\ \text{v0.y} \\ \text{v0.z} \\ \text{v0.w} \end{bmatrix}$$

m4x4는 또한 다음과 같이 4번의 내적(dp4: 4차원 벡터의 내적)를 수행하는 것과 같습니다.

$$\begin{array}{ll} \text{m4x4 r0, r1, c0} \Leftrightarrow & \begin{array}{l} \text{dp4 r0.x, r1, c0} \\ \text{dp4 r0.y, r1, c1} \\ \text{dp4 r0.z, r1, c2} \\ \text{dp4 r0.w, r1, c3} \end{array} \end{array}$$

m4x4 외에 m4x3, m3x3도 있는데 m3x3은 주어진 행렬의 3행 3열만 곱셈 연산에 적용이 됩니다.

c0에 "월드 행렬 \* 뷰 행렬 \* 투영 행렬" 값을 전달하기 위해서 [s0v\\_02\\_shader\\_file.zip](#)의 CShaderEx::Render() 함수를 보면 다음과 같이 행렬 값을 설정하고 SetVertexShaderConstant() 함수로 상수 레지스터 c0에 설정하고 있음을 볼 수 있습니다.

```

D3DXMATRIX      mtWld;           // 월드 행렬
D3DXMATRIX      mtViw;           // 뷰 행렬
D3DXMATRIX      mtPrj;           // 투영 행렬
...
D3DXMATRIX      mtVP    = mtWld * mtViw * mtPrj;
D3DXMatrixTranspose( &mtVP , &mtVP );
...
m_pDev->SetVertexShaderConstantF( 0, (FLOAT*)&mtVP , 4);

```

지금은 정점 변환에 관한 행렬을 전부 곱해서 전달하고 있고 각각 월드 변환, 뷰 변환, 정규 변환을 쉐이더에서 단계적으로 구현하기 위해서 먼저 다음과 같이 순서와 수식을 정리합니다.

정점의 위치' = 입력 정점의 위치

정점의 월드 변환 위치' = 정점의 위치' \* 월드 행렬

정점의 뷰 변환 위치' = 정점의 월드 변환 위치' \* 뷰 행렬

정점의 정규 변환 위치' = 정점의 월드 변환 위치' \* 투영 행렬

이것을 쉐이더 코드로 전환해야 하는데 월드 행렬, 뷰 행렬, 투영 행렬은 외부에서 설정되는 값이므로 상수 레지스터로 설정합니다. 그런데 행렬은 float형 16개가 필요하기 때문에 만약 월드 행렬을 c0에 설정하게 되면 c0~c3까지 상수 레지스터가 월드 행렬 값으로 설정이 되고, 뷰 행렬이 c4에 설정되면 c4~c7 레지스터가 뷰 행렬 값으로 설정 됩니다. 투영 행렬이 c8에 설정되면 c8~c11까지 투영 행렬 값이 설정 됩니다.

수식의 왼쪽에 있는 정점의 위치', 정점의 월드 변환 위치, 뷰 변환 위치, 정규 변환 위치는 쉐이더 내부에서 임시(Temporary)로 사용되는 값입니다. 따라서 이들은 임시 레지스터 r0, r1, r2, r3에 저장 시킬 수 있습니다.

이 내용을 가지고 저 수준 쉐이더 코드로 전환하면 다음과 같습니다.

```
vs_1_1           // 쉐이더 버전
dcl_position    v0   // 위치 설정
mov r0, v0       // 정점의 위치를 임시 레지스터에 복사
m4x4 r1, r0, c0 // 월드 행렬 변환 후 r1에 저장
m4x4 r2, r1, c4 // 뷰 행렬 변환 후 r2에 저장
m4x4 r3, r2, c8 // 투영 행렬 변환 후 r3에 저장
mov oPos, r3     // r3 값을 출력 레지스터 oPos에 복사
```

만약 정점의 색상을 출력하려면 "dcl\_color" 또는 "dcl\_color0"를 이용해서 정점의 Diffuse 값에 대해서 레지스터를 선언합니다. 만약 정점의 스페큘러(Specular) 값을 얻으려면 "dcl\_color1"을 이용해야 합니다.

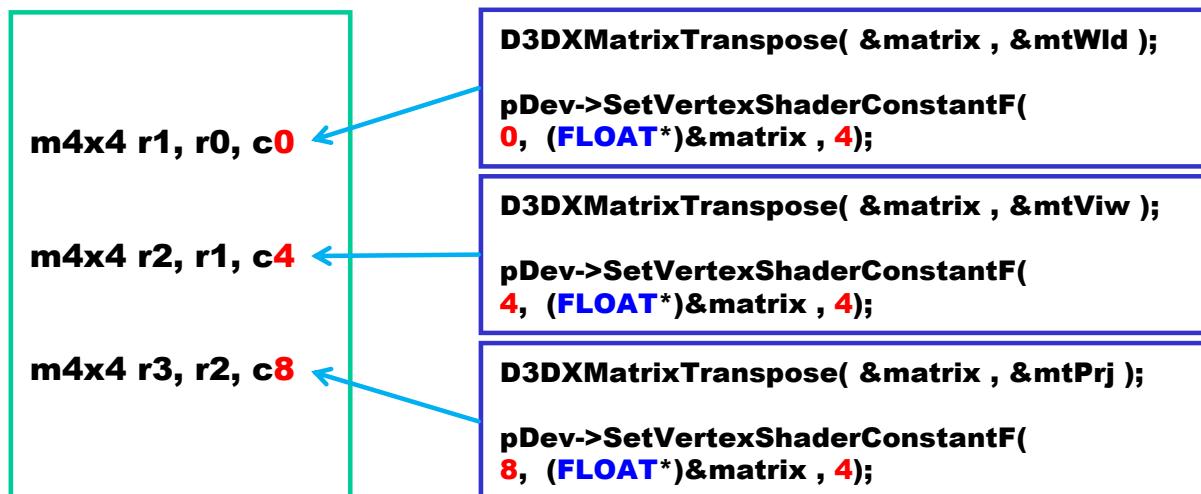
```
dcl_color      v1   // 색상 레지스터
...
mov oD0, v1     // 정점의 색상을 직접 복사
```

색상을 출력하기 위해서는 oD0와 oD1을 사용합니다. oD0는 Diffuse 값, oD1은 스페큘러 값에 대한 출력 레지스터입니다.

상수 레지스터 c0, c4, c8에 값을 설정하기 위해서 우리는 다음과 같이 먼저 행렬을 전치(Transpose)시키고, 디바이스의 SetVertexShaderConstantF() 함수에 "상수 레지스터 인덱스", 행렬 주소, 4 등을 인수로 전달하면 됩니다.

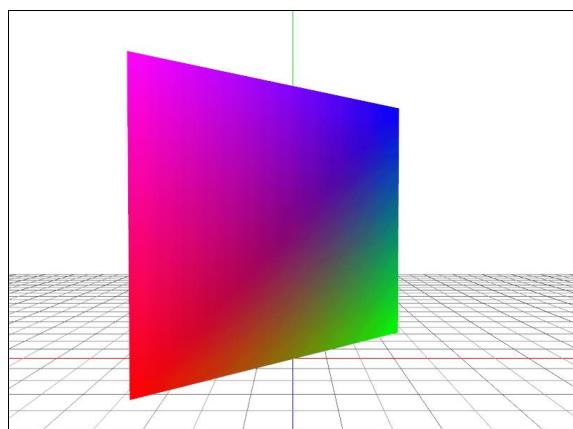
```
D3DXMATRIX      mtT;
D3DXMatrixTranspose( &mtT , &mtWld );
m_pDev->SetVertexShaderConstantF( 0, (FLOAT*)&mtT , 4 );
D3DXMatrixTranspose( &mtT , &mtViw );
```

```
m_pDev->SetVertexShaderConstantF( 4, (FLOAT*)&mtT , 4);
D3DXMatrixTranspose( &mtT , &mtPrj );
m_pDev->SetVertexShaderConstantF( 8, (FLOAT*)&mtT , 4);
```



<SetVertexShaderConstantF() 함수의 인덱스와 상수 레지스터 관계>

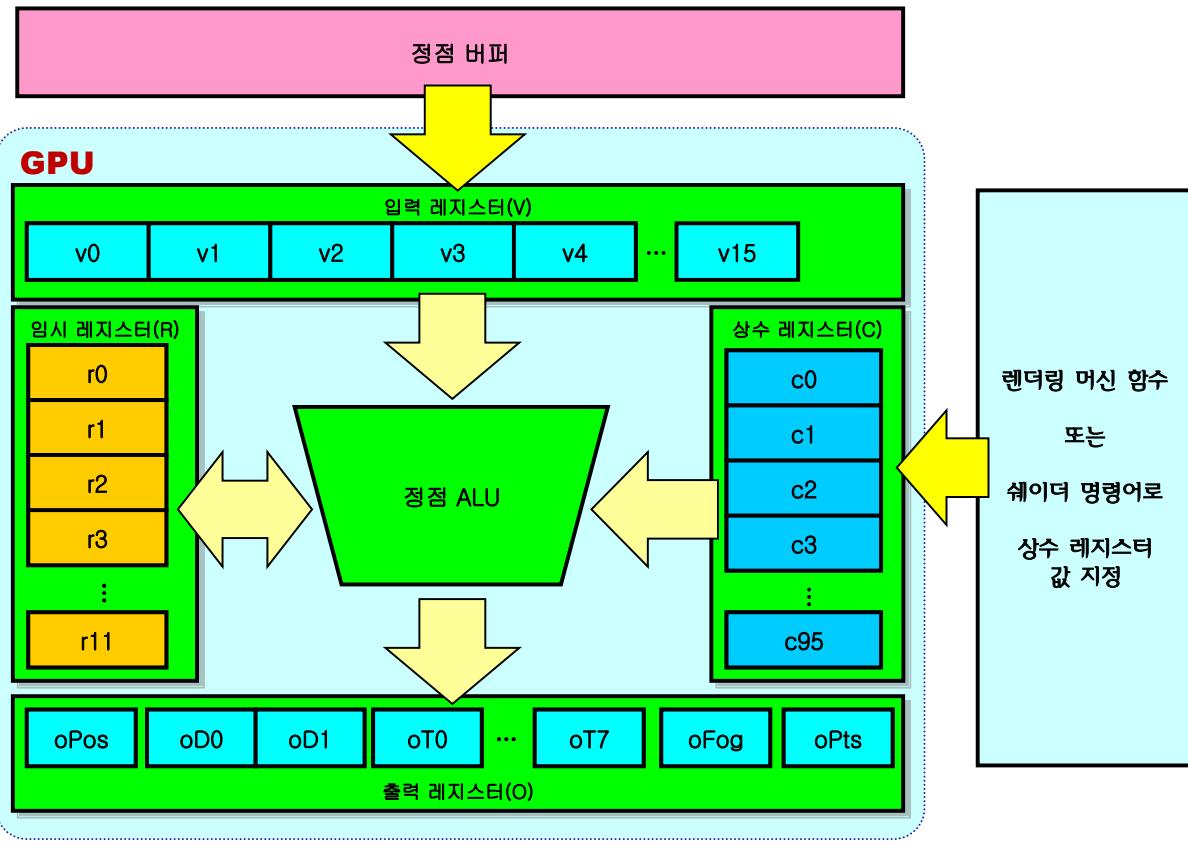
[s0v\\_04\\_transform.zip](#)은 정점의 월드 변환, 뷰 변환, 정규 변환을 저 수준으로 구현한 예제입니다.



<정점의 변환: [s0v\\_04\\_transform.zip](#)>

## 2.5 Vertex Shader Virtual Machine

쉐이더가 익숙해지기 위해서 정점 쉐이더의 레이아웃을 알아보았고 정점의 변환을 연습해보았습니다. 정신 없이 쉐이더 강의를 진행 했는데 다음 단계를 위해서 차분하게 정점 쉐이더의 구조를 다시 살펴 보도록 합시다.



&lt;정점 쉐이더 가상 머신&gt;

그림은 정점 쉐이더 가상 머신(Virtual Machine)을 표현한 것입니다. 이 가상 머신은 크게 입력 레지스터, 출력 레지스터, 상수 레지스터, 임시 레지스터, 그리고 산술과 논리 연산을 담당하는 ALU로 구성되어 있습니다.

보통 마이크로 프로세서 내부에서 처리를 돋기 위해 만들어진 작은 기억 공간을 레지스터라 합니다. GPU도 하나의 "처리기(Processor)" 이므로 GPU 내부에서 처리를 보조하는 기억 공간도 레지스터라 부르는 것입니다.

ALU는 Arithmetic-Logic Unit로 정점 ALU는 정점 데이터에 대한 산술 연산과 논리 동작을 담당합니다. ALU는 기본적인 사칙 연산부터 벡터 \* 행렬의 연산, 내적, 제곱근, 승수, exp, log 등의 수학 함수들과 조건 문 등을 처리할 수 있습니다. 정점 쉐이더 코드를 작성한다는 점을 단순하게 바라본다면 정점 ALU가 처리하는 과정을 순서대로 나열한 것이라 할 수 있습니다.

그림을 보면 정점 쉐이더 가상머신의 흐름이 노란색 화살표로 표시되어 있고 입력된 정점 버퍼의 데이터는 입력 레지스터에 "dcl\_" 문장에 의해서 정점의 위치, 법선, 색상, 텍스처 좌표 등등이 각각 분리되어 입력 레지스터 v0~v15에 적재 됩니다.

정점 ALU는 입력 레지스터의 값을 받아서 연산을 하며 필요하다면 상수 레지스터와 임시 레지스터

를 사용하기도 합니다.

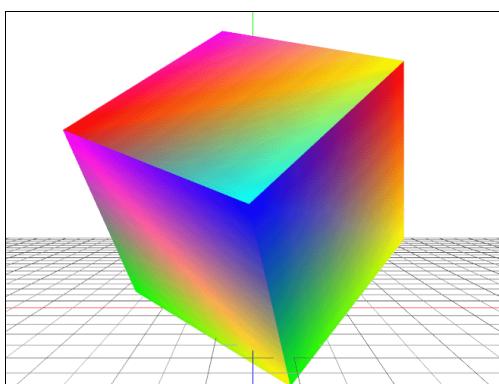
우리가 정점 쉐이더를 컴파일하고 파이프라인에 적용하게 되면 동작을 바꾸어 볼 수 있는 것은 정점 베피와 상수 레지스터뿐이라는 것을 기억해야 합니다. 따라서 다양한 기능을 정점 쉐이더에 적용하려면 많은 변수들을 상수 레지스터로 구성해야 합니다. 이로 인해 GPU 내부에서 가장 큰 레지스터를 구성하는 것이 상수 레지스터입니다.

상수 레지스터의 값은 "def" 문장으로 쉐이더 코드 내부에서 지정하거나 아니면 외부에서 SetVertexShaderConstant{F|B|I}() 함수로 설정합니다. 같은 레지스터의 값을 지정할 때는 "def"로 명명한 우선 순위가 쉐이더 내부에서 정한 값이 먼저입니다.

우리는 연산의 과정에서 일시적으로 값을 저장할 필요가 있습니다. 이 때 임시 레지스터를 이용합니다. 입력 레지스터와 상수 레지스터는 읽기 전용 이지만 임시 레지스터는 읽기/쓰기가 가능한 레지스터이며 임시 레지스터에 값을 설정할 때는 "mov" 명령어를 이용해야 합니다.

정점에 관한 모든 처리를 끝냈으면 출력 레지스터에 결과를 저장해야 합니다. 출력 레지스터는 소문자 "o"(Out)로 시작을 하는 레지스터로 위치, 색상, 텍스처 좌표, 안개, 점 크기 5종류가 있습니다. 출력 레지스터의 위치는 oPos에 저장하고, 색상에 대한 Diffuse와 Specular 값은 oD0, oD1에 저장합니다. 텍스처 좌표는 oT0~oT7까지 사용할 수 있습니다. 안개는 oFog, Point의 크기는 oPts에 저장합니다. 때로는 픽셀 쉐이더에서 정점 쉐이더의 입력 값 또는 결과 값을 사용하고자 할 때가 있습니다. 프로그래머들은 이 경우에 출력 레지스터 중에서 텍스처 좌표를 저장하는 oT0~oT7을 사용하며 끝 번호(oT7)부터 주로 이용합니다.

```
dcl_position    v0      // 정점 위치
dcl_normal     v1      // 정점 법선
...
mov  oT7, v0          // 정점 위치를 oT7에 저장
mov  oT6, v1          // 정점 법선 벡터를 oT6에 저장
```



<상수 레지스터, 임시 레지스터, 출력 레지스터 사용 예. [s0v\\_05\\_const.zip](#)>

GPU 레지스터의 크기는 float \* 4로 구성되어 있습니다. 또한 하나의 레지스터는 x, y, z, w 와 r, g, b, a를 구분하지 않고 x, y, z, w 순서 또는 r, g, b, a 순으로 저장 됩니다. 이것은 색상도 벡터에서 적용되는 연산이 가능하며 또한 벡터도 색상으로 출력할 수 있다는 의미도 됩니다.

쉐이더는 각 버전마다 전체 명령에 대한 제한이 있습니다. 따라서 필요한 명령어 이외의 쉐이더 코드는 GPU의 불필요한 동작을 지시하는 것이기 때문에 쉐이더 작성에서 한 줄이라도 명령문을 줄이는 것이 좋습니다. [s0v\\_05\\_const.zip](#)의 "data/ shader.vsh" 은 강의를 목적으로 작성된 것이므로 이렇게 작성하는 것은 좋지 않으며 가급적이면 필요한 부분만 남겨 놓거나 줄여서 사용하는 것이 바람직합니다.

지금까지 정점의 변환과 색상을 가지고 프로그램을 작성해 보았는데 텍스처, 조명등을 살펴보겠습니다.

## 2.6 Texture Mapping

텍스처에 관련된 중심 내용은 픽셀 쉐이더에 있습니다. 보통 정점 쉐이더에서 텍스처 적용은 단순히 좌표 전달 정도의 용도로 활용 되는 것이 대부분입니다. 이 경우에 쉐이더 코드는 다음과 같이 작성합니다.

1. 정점의 텍스처 좌표에 대해서 "dcl\_texcoord"로 입력 레지스터의 텍스처 좌표를 설정한다.
2. 필요에 따라 텍스처 좌표를 변환한다.
3. 출력 레지스터 oT0~oT7에 복사한다.

입력 레지스터에 대한 정점의 텍스처 좌표 설정은 다음 예제와 같이 dcl\_texcoord 명령어로 설정 합니다.

`dcl_texcoord v3`

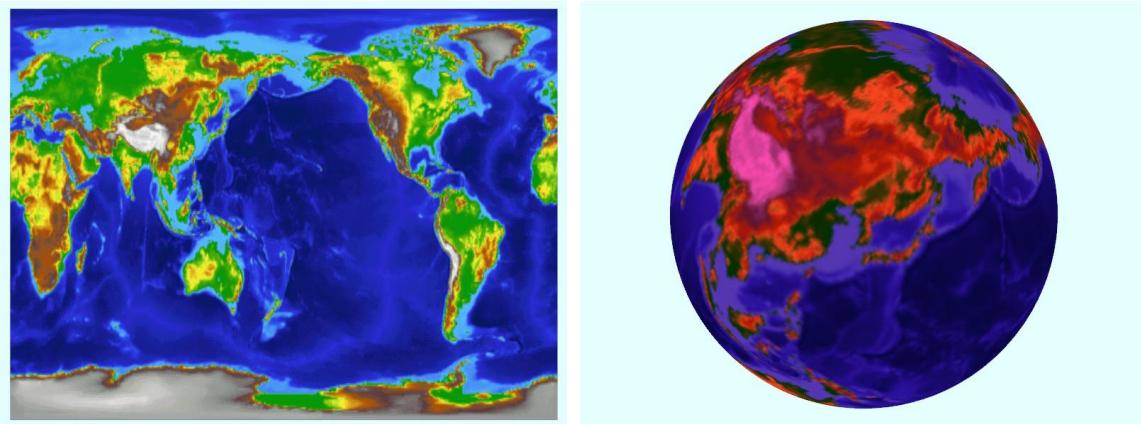
입력 텍스처 좌표는 총 8개까지 설정이 가능합니다. 여러 개의 텍스처 좌표를 입력 레지스터에 설정 할 때는 dcl\_texcoord0~ dcltexcoord7을 사용합니다. dcl\_texcoord는 dcl\_texcoord0과 같습니다.

텍스처 좌표도 위치이므로 정점의 위치와 같이 변환(Transform)을 할 수 있습니다. 여기서는 텍스처 좌표의 변환이 없다고 가정하고 최종 단계인 출력 레지스터에 직접 복사하도록 합시다.

`mov oT0, v3`

$oT0$ 는 Output Texture 0-address coordinate를 의미하며 DirectX는 총 8장의 멀티 텍스처를 하나의 그래픽 파이프라인에서 사용이 가능하므로 사용자는  $oT0 \sim oT7$ 까지 출력 레지스터에 복사를 할 수 있습니다.

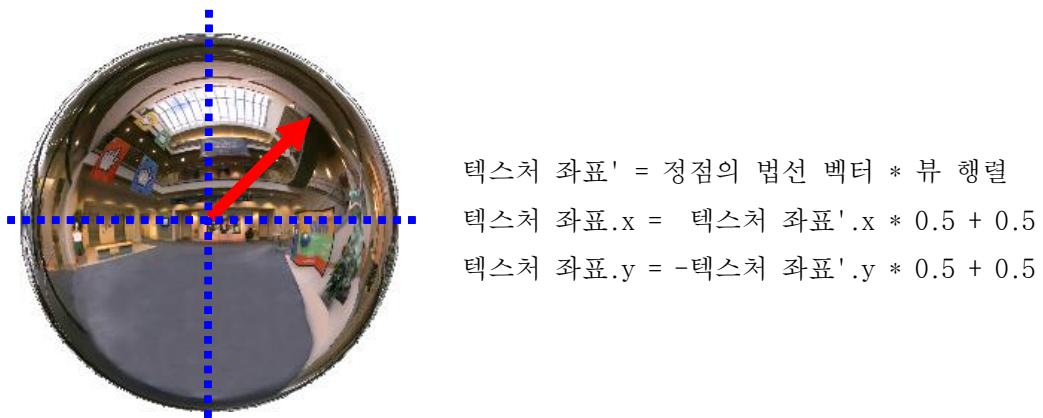
[s0v\\_06\\_tex.zip](#)와 [s0v\\_06\\_tex\\_earth.zip](#)의 "data/shader.vsh"는 은 단순히 텍스처 좌표를 출력 레지스터로 복사하는 예제입니다.



<텍스처 출력. [s0v\\_06\\_tex.zip](#), [s0v\\_06\\_tex\\_earth.zip](#)>

반사 또는 굴절에 대한 환경 매핑(Environment Mapping)은 정점의 좌표와 법선 벡터를 텍스처 좌표로 전환하는 것입니다. 이 내용은 3D 기초 시간에 약간의 복잡한 과정을 거쳐 살펴보았는데 환경 매핑은 쉐이더를 사용하면 쉽게 구현할 수 있습니다.

반사라는 것은 다음과 같이 주어진 그림에서 붉은 색 화살표의 텍스처 좌표를 설정하는 것이고, 이 붉은 색 화살표는 정점의 법선 벡터를 이용하면 됩니다.



또한 반사는 카메라의 위치에 의해 결정되기 때문에 정점의 법선 벡터를 카메라 공간으로 변환하

면 카메라의 움직임에 대해서도 반사효과를 만들어 낼 수 있습니다. 이것을 수식으로 정리하면 그림 옆에 있는 수식과 같습니다.

수식에서 텍스처 좌표에 0.5를 곱하고 더한 것은 법선 벡터의 범위가 [-1, 1]이기 때문에 텍스처 중심으로 좌표를 [0, 1] 범위로 만들기 위해서입니다. 이 수식을 쉐이더로 구현하면 다음과 같습니다.

```
#define Nor      r0
#define Tex       r1
...
def      c32, 0.4, 0.5, 0.0, 1.0
dcl_normal     v1           // 정점의 법선 벡터 레지스터 선언
...
m3x3 Nor, v1,   c4           // 법선 벡터의 카메라 공간 변환
mov Tex.xyz, Nor.xyz          // 변환된 법선 벡터를 텍스처 좌표에 복사
mov Tex.w, c32.w              // 텍스처 좌표의 w 값 설정
mov Tex.y, -Tex.y             // y 좌표를 반전시킴

// Tex = Tex * 0.4F + 0.5F
mad Tex, Tex, c32.x, c32.y      // 텍스처의 중심으로 이동과 범위[0,1]로 설정
mov oT0, Tex
```

저 수준 쉐이더에서는 전처리 문이 지원이 되며 "#define" 키워드로 매크로를 정의할 수 있습니다. 카메라 움직임에 대해서 반사 효과가 적용되어야 하기 때문에 정점의 법선 벡터를 뷰 행렬에 곱합니다. 이 때 법선 벡터의 크기는 변하지 않아야 하므로 회전 변환만 적용시키기 위해서 m3x3을 사용합니다.

법선 벡터가 mad를 이용해서 텍스처 중심 좌표 (0.5, 0.5)에서 [0, 1] 범위로 이동하는 계산을 한번에 처리하기 위해서 y 값을 반전시켜 놓습니다. mad 명령어는 다음과 같이 두 변수의 곱에 세 번째 변수를 합한 값을입니다.

	r3.x = r0.x * r1.x + r2.x;
	r3.y = r0.y * r1.y + r2.y;
mad r3, r0, r1, r2	$\equiv$
	r3.z = r0.z * r1.z + r2.z;
	r3.w = r0.w * r1.w + r2.w;

따라서 "mad Tex, Tex, c32.x, c32.y"는 다음과 같은 의미이며 여기서 0.5F 대신 0.4F를 사용한

것은 환경 매핑의 경계에서 해상도가 낮기 때문입니다.

$\text{Tex.x} = \text{Tex.x} * 0.4F + 0.5F, \text{ Tex.y} = \text{Tex.y} * 0.4F + 0.5F, \dots$

이 내용은 [s0v\\_06\\_tex\\_env.zip](#)의 "data/shader.vsh" 파일에 구현되어 있습니다. 실행하면 다음과 같은 화면을 볼 수 있습니다.



<환경 매핑: [s0v\\_06\\_tex\\_env.zip](#), 정점 위치로 구현된 텍스처 좌표: [s0v\\_06\\_tex\\_vtx.zip](#)>

3D 기초에서 정점 좌표를 텍스처 좌표로 사용한 예도 있었는데 구현이 간단하므로 [s0v\\_06\\_tex\\_vtx.zip](#) 예제를 참고 하기 바랍니다.

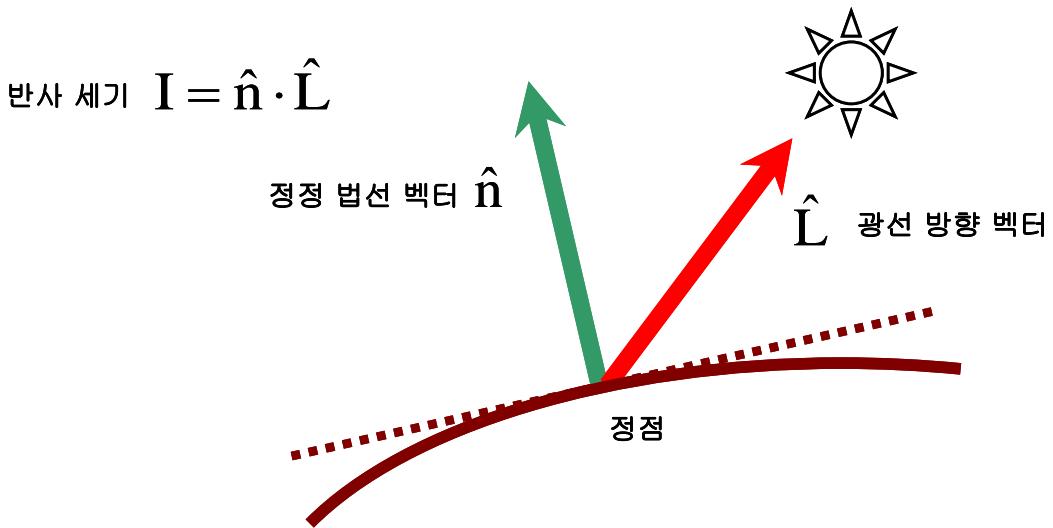
## 2.7 Lighting

지금까지 정점 쉐이더에서 변환, 색상, 텍스처 좌표 설정 등을 살펴 보았습니다. 정점 쉐이더에서 가장 비중이 있는 영역은 조명(Lighting)입니다. 특히, 조명 원리를 잘 알고 있으면 NPR(Non Photo-realistic Rendering)의 대표인 Toon(Cartoon) Shading을 쉽게 구현할 수 있습니다.

조명 효과를 구현하기 위한 모델은 여러 가지가 있습니다. 이 중에서 우리는 3D 기초 시간에 다루었던 Lambert 확산과 Phong 반사를 각각 쉐이더로 구현해 보겠습니다. 다음으로 Phong 반사를 개선한 Blinn-Phong 반사를 알아보고 Lambert 확산과 Blinn-Phong 반사를 동시에 렌더링에 적용해 보도록 하겠습니다.

### 2.7.1 Lambert 확산

Lambert 확산은 그림과 같이 정점의 법선 벡터와 빛의 방향 벡터의 내적을 반사의 세기 (Intensity)로 사용하는 것입니다.



<Lambert 확산(반사)의 세기>

렌더링 오브젝트는 월드 변환할 수 있습니다. 월드 변환에서 법선 벡터는 회전만 적용해야 합니다. 회전을 법선벡터에 적용하려면 월드 행렬이 크기 변환이 없는 경우에 대해서 "m3x3"을 이용합니다 m3x3은 주어진 행렬의 3행 3열만 연산에 적용됩니다.

월드 변환을 하는 오브젝트에 대한 반사의 세기를 쉐이더 코드로 작성하면 다음과 같습니다.

```
#define Nor      r0
#define Lgt      -c8

vs_1_1
...
dcl_normal      v1      // 정점 법선 벡터를 입력 레지스터 v1에 선언
...
m3x3 Nor, v1, c4      // 법선 벡터는 회전만 적용
dp3 r1.w, Nor, Lgt      // Light 방향과 내적으로 정점의 밝기를 설정
```

dp3는 Dot Product(내적)으로 "dp" 다음의 숫자는 차원을 나타냅니다. 3이면 xyz만 수행하고 4면 xyzw 성분에 대해서 내적을 구합니다. 만약 "dp3 r2, r0, r1"와 같이 결과를 저장하는 장소를 명시하지 않으면 dp3는 r2의 xyz 성분에 같은 내적 값을 기록합니다. 앞의 코드는 내적의 결과를 r1.w에 저장하고 있습니다.

반사의 세기 I는 내적에 의해 범위가 [-1, 1] 가 됩니다. 그런데 밝기는 (-)가 없으므로 0보다 작으면 0으로 만드는 Saturation을 사용하거나 아니면 전체 밝기에 1을 더한 다음 다시 0.5를 곱해

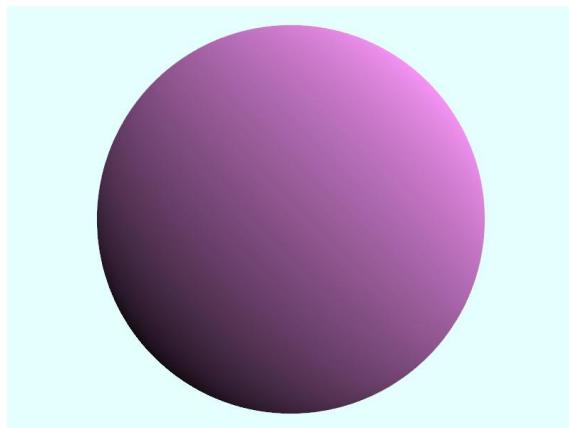
서 [0, 1] 범위로 조정할 수 있습니다. 여기서 전체 밝기에 1을 더하고 0.5를 곱하는 것을 적용해봅시다.

" $I = (I + 1)/2$ "에 대해서 쉐이더를 사용하면 add와 mul 연산이 필요합니다. 그런데 mad를 사용하면 한 번에 처리가 됩니다. " $I = (I + 1)/2$ "는 " $I * 0.5 + 0.5$ "와 같으므로 다음과 같이 mad를 이용해서 쉐이더 코드를 작성합니다.

```
def c24, 1, 0.5, 0.1, 1. 0
...
mad      r1, r1.w, c24.y, c24.y
```

반사의 세기를 계산했습니다. 만약 광원의 색상이 있으면 이 값을 반사의 세기에 곱한 후에 출력레지스터 ODO에 복사합니다.

```
mov      r1.w, c24.w
mul      oDO, r1, c10    // 최종 색상 = Lambert 반사 * 빛의 색상
```

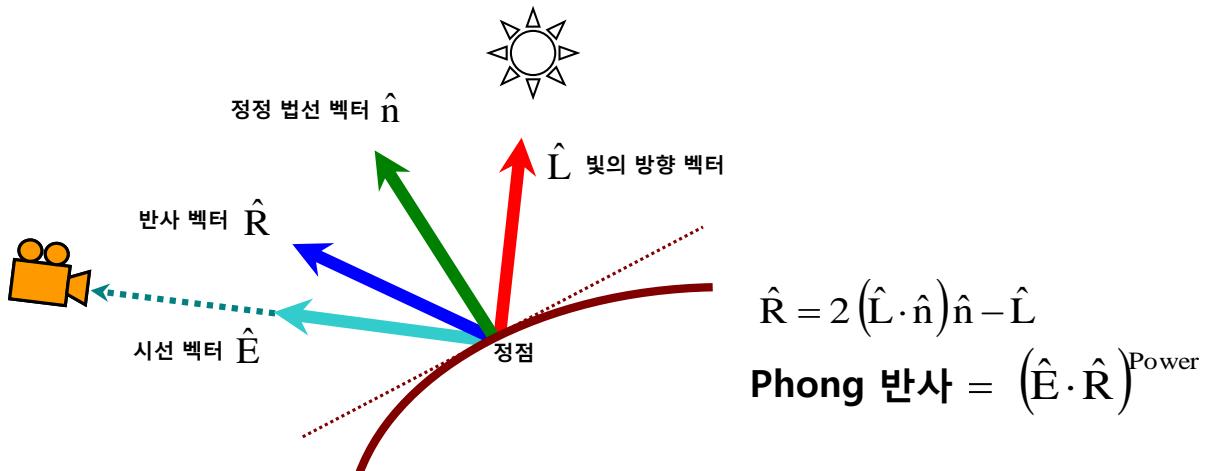


<Lambert 확산. [s0v\\_07\\_llambert.zip](#)>

## 2.7.2 Phong 반사

퐁(Phong) 반사는 렌더링 오브젝트의 정 반사(Specular) 효과를 표현한 조명 모델입니다. 퐁 반사의 세기는 정점에서 카메라의 위치에 대한 시선 벡터와 반사 벡터의 내적에 멱승 (Power)을 적용해서 구합니다.

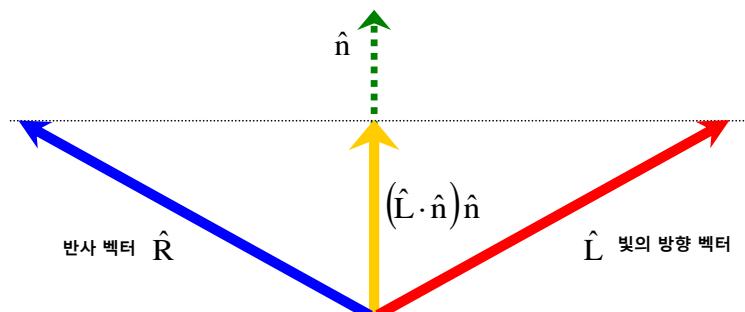
렌더링 오브젝트는 크기, 회전, 이동에 대한 월드 변환이 적용되므로 시선 벡터를 구하기 전에 먼저 정점의 위치에 대해서 월드 변환을 적용하며, 법선 벡터는 Lambert 확산에서처럼 회전만 적용합니다.



<퐁 반사 모델>

반사 벡터는 빛의 방향 벡터와 정점의 법선 벡터를 이용해서 구합니다. 반사 벡터를 빠르게 구하는 방법은 먼저 법선 벡터와 빛의 방향 벡터를 이용해서 법선 벡터 방향으로  $(\hat{L} \cdot \hat{n})\hat{n}$ 을 만듭니다.

반사 벡터  $\hat{R}$  과  $\hat{L}$ 을 더하면  $2 * (\hat{L} \cdot \hat{n})\hat{n}$ 이 되므로  $\hat{R}$ 을 쉽게 구할 수 있습니다.



<반사 벡터 계산>

$$\hat{R} + \hat{L} = 2 * (\hat{L} \cdot \hat{n})\hat{n} \quad \therefore \hat{R} = 2 * (\hat{L} \cdot \hat{n})\hat{n} - \hat{L}$$

퐁 반사를 저 수준 쉐이더 언어로 직접 작성하는 것은 쉬운 일이 아닙니다. 좀 더 편리한 방법은 다음과 같은 의사(Pseudo) 코드를 먼저 작성하고 이것을 쉐이더 언어로 변경하는 것입니다.

변환 위치 = 정점 위치 \* 월드 행렬

변환 법선 = 정점 법선 \* 월드 행렬의 회전 행렬

```
시선 벡터 = normalize(카메라 위치 - 변환 위치)
반사 벡터 = 2 * dot(빛의 방향, 변환 법선) * 변환 법선 - 빛의 방향
퐁 반사의 세기 = dot(시선 벡터, 반사 벡터)^Power
```

쉐이더 코드 작성을 편리하게 하기 위해서 다음과 같이 전 처리문을 사용해서 레지스터 이름을 정의합니다.

```
#define Pos      r0
#define Nor      r1
#define Eye      r2
#define Rfc      r3
#define Phn      r4
#define Lgt      -c8
#define Cam      c16
#define Pow      c16.w
```

'변환 위치' 월드 변환 행렬을 그대로 적용하면 되지만 '변환 법선'은 회전만 적용해야 합니다. 만약 크기 변환 행렬이 적용 안된 월드 변환 행렬이라면 '변환 법선'은 회전만 적용하면 되므로 m3x3으로 '변환 법선'을 구할 수 있습니다. 다음은 '변환 위치', '변환 법선'을 구하는 쉐이더 코드입니다.

```
dcl_position    v0
dcl_normal      v1
...
m4x4 Pos, v0, c4          // 월드 변환
m3x3 Nor, v1, c4          // 법선 벡터는 회전만 적용
```

시선 벡터는 normalize(카메라의 위치 - 변환 위치) 입니다. 먼저 카메라의 위치에서 변환 위치를 뺍니다.

```
sub Eye, Cam, Pos
```

시선 벡터를 단위 벡터로 만들어야 하는데 일반 벡터를 단위 벡터로 만드는 방법은 자신의 크기로 나누는 것입니다.

$$\hat{\vec{v}} = \frac{\vec{v}}{|\vec{v}|}$$

저 수준 쉐이더에서 rsq는 입력된 t의 크기의 역수  $1/\sqrt{t}$  를 반환하는 연산자입니다.

$1/|\vec{v}|$  는  $1/\sqrt{\vec{v} \bullet \vec{v}}$  와 동등 하므로 먼저 시선 벡터를 dp3로 내적을 구하고 rsq를 사용하면  $1/(벡터 크기)$ 와 동등해집니다. 이 값을 다시 시선 벡터에 곱하면 시선 벡터가 단위 벡터가 됩니다.

dp3 Eye.w, Eye, Eye

rsq Eye.w, Eye.w

mul Eye.xyz, Eye.xyz, Eye.www

"반사 벡터 =  $2 * \text{dot}(\text{빛의 방향}, \text{변환 법선}) * \text{변환 법선} - \text{빛의 방향}$ " 를 구하기 위해서 변환 법선과 빛의 방향 벡터의 내적을 먼저 구합니다. 내적 값에 2를 곱해야 하는데 같은 내적 값을 add를 하면 2를 곱한 결과와 동일합니다. 다시 변환 법선 벡터를 곱한 다음 빛의 방향 벡터를 빼주면 반사 벡터를 구하게 됩니다.

```
dp3 Rfc.w, Nor, Lgt           // dot(N, L)
add Rfc.w, Rfc.w, Rfc.w       // * 2
mul Rfc.xyz, Nor, Rfc.www     // * N
sub Rfc, Rfc, Lgt             // - L
```

시선 벡터, 반사 벡터를 구했으니 풍 반사의 세기를 구할 수 있습니다.

```
dp3 Phn.w, Eye, Rfc           // dot(E, R)
```

풍 반사의 세기도 내적이므로 Lambert 때와 마찬가지로 [0, 1] 범위로 조정합니다.

```
def    c24, 1, .5, 0.1, 1.
...
add    Phn.w, Phn.w, c24.x    // limit [0, 1]
mul    Phn.w, Phn.w, c24.y
```

마지막으로 역승(Power)을 적용합니다. 쉐이더 2.0 이후에는 pow 연산자가 있지만 1.1은 없기 때문에 다음과 같은 수식을 exp와 log 함수로 만듭니다.

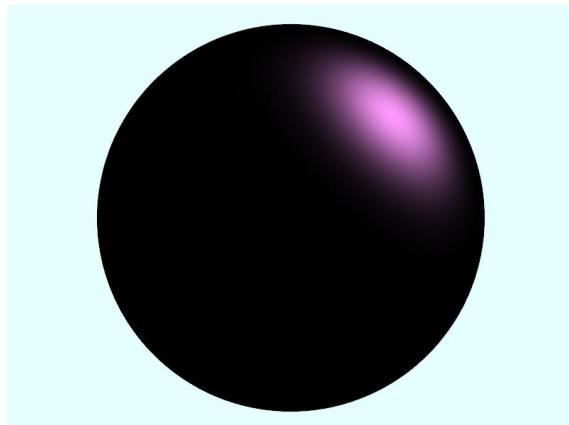
$$\text{Power}(A, B) = A^B = e^{\log A^B} = \exp(\log A^B) = \exp(B * \log A)$$

B 대신 외부에서 Specular의 Power에 대한 Pow 변수로 바꾸고 A 대신 내적을 적용하면  
 $\log(\dot{E}, R) * \text{Pow}$  가 되어 다음과 같은 쉐이더 코드를 만들 수 있습니다.

```
log    Phn.w, Phn.w
mul    Phn.w, Phn.w, Pow
exp    Phn.w, Phn.w
```

마지막으로 빛의 색상을 곱하고 풍 반사의 w 값을 1로 설정한 다음 oD0 레지스터에 복사합니다.

```
mul    Phn, c10, Phn.w      // Color = 풍 반사 * 빛의 색상
mov    Phn.w, c24.w        // Color.w = 1.f;
mov    oD0, Phn            // Output Diffuse Color
```

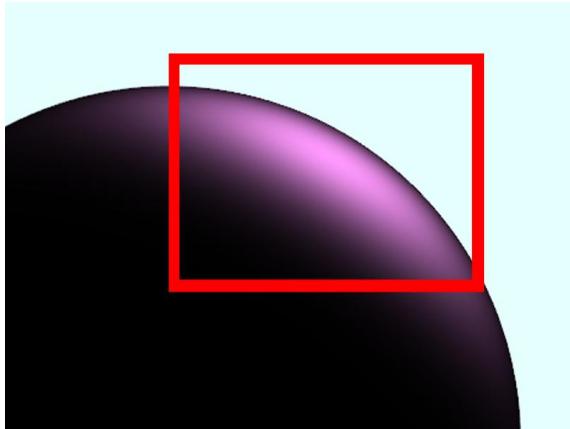


<풍 반사: [s0v\\_07\\_2phong.zip](#)>

### 2.7.3 Blinn-Phong 반사

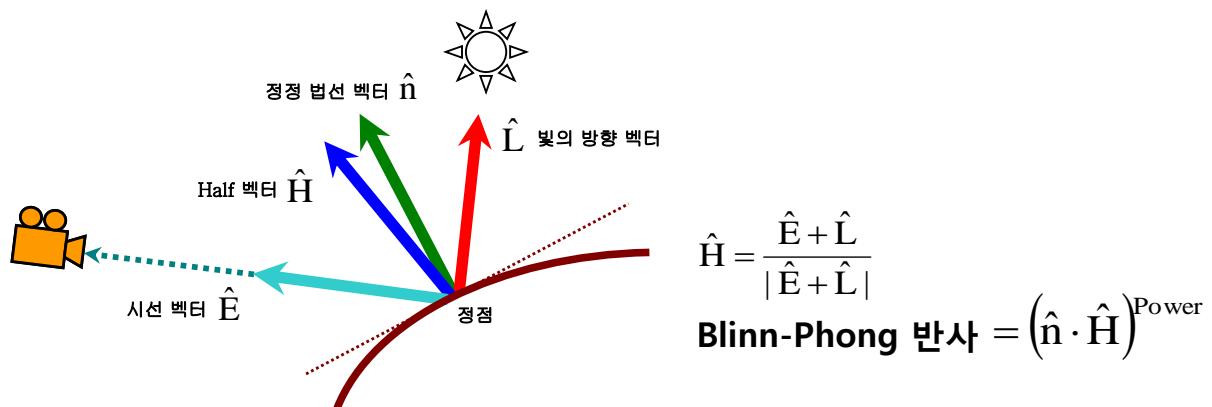
풍 반사는 정면으로 반사하는 빛에 대해서 현실 세계를 잘 표현하지만 다음 그림과 같이 거의 수평면으로 입사되는 빛에 대해서 더 넓은 영역의 하이라이트(Highlight)를 만들고 반사의 경계를 만들어 냅니다.

또한 실 세계에서 거의 수평면으로 입사한 빛은 오히려 더 강한 Specular를 만들고, 이를 표현하려면 시선 벡터 방향으로 반사 벡터를 좀 더 움직여야 합니다. 이것을 "off-specular peak" 이라 합니다.



&lt;거의 수평으로 입사한 빛의 풍 반사의 off-specular peak&gt;

Blinn-Phong 반사는 풍 반사 모델을 수정해서 좀 더 현실 세계의 정 반사 효과를 표현한 조명 모델이라 할 수 있습니다. Blinn-Phong 반사는 다음 그림과 같이 풍 반사의 반사 벡터 대신 Half 벡터를 사용합니다.



&lt;Blinn-Phong 반사 모델&gt;

Half 벡터는 시선 벡터와 빛의 방향 벡터의 합으로 다음과 같이 간단하게 계산합니다.

$$\hat{H} = \frac{\hat{E} + \hat{L}}{|\hat{E} + \hat{L}|}$$

Blinn-Phong 반사 세기를 구하는 과정은 풍 반사에서 반사 벡터 대신 Half 벡터를 구하고, Half와 변환된 법선 벡터의 내적을 반사의 세기로 설정합니다.

변환 위치 = 정점 위치 \* 월드 행렬

변환 법선 = 정점 법선 \* 월드 행렬의 회전 행렬

시선 벡터 = normalize(카메라 위치 - 변환 위치)

Half 벡터 = normalize(시선 벡터 + 빛의 방향 벡터)

Blinn-Phong 반사 세기 = dot(법선 벡터, Half 벡터)^Power

다음은 시선 벡터와 빛의 방향 벡터를 이용해서 Half를 구하는 쉐이더 코드입니다.

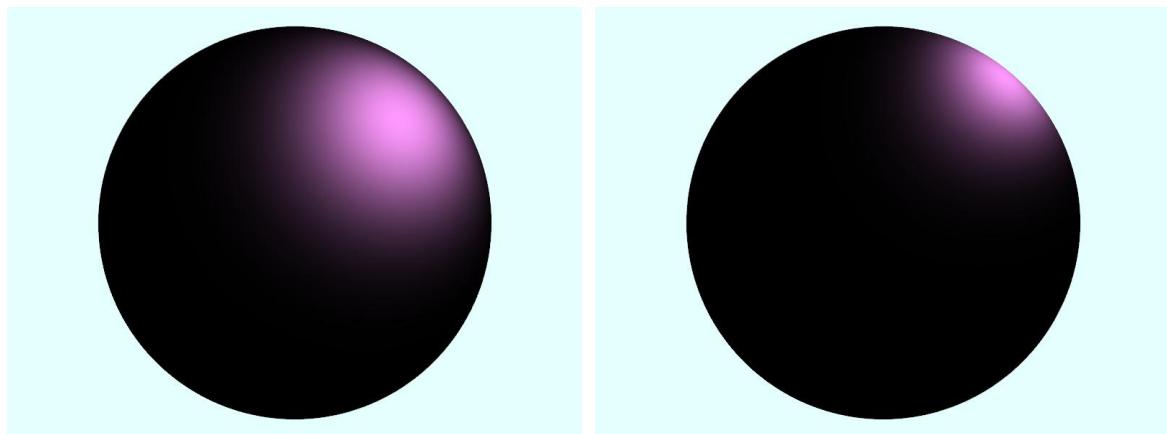
```
add Hlf, Eye, Lgt           // H = E + L
dp3 Hlf.w, Hlf, Hlf         // Normalize H
rsq Hlf.w, Hlf.w
mul Hlf.xyz, Hlf.xyz, Hlf.www
```

이후 Blinn-Phong 반사 세기를 구하는 내용은 풍 쉐이딩에서 Half 벡터와 변환된 법선 벡터의 내적만 다르고 나머지는 같습니다.

```
dp3    Bln.w, Hlf, Nor      // dot(H, N)
...
log    Bln.w, Bln.w        // pow(Blinn, Power)
mul    Bln.w, Bln.w, Pow
exp    Bln.w, Bln.w

mul    Bln, c10, Bln.w     // Blinn-Phong 반사 * 빛의 색상
mov    Bln.w, c24.w        // Color.w = 1.f;
mov    oD0, Bln            // Output Diffuse Color
```

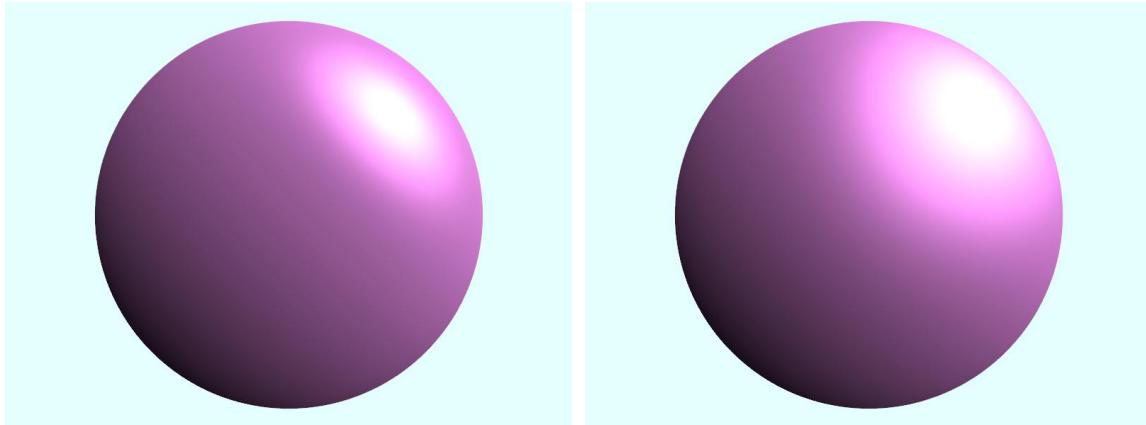
Blinn-Phong 반사는 풍 반사보다 반사의 영역이 넓지만 그림과 같이 거의 수평으로 반사되는 빛에 대해서도 잘 표현 됩니다. 반사 영역을 좁히는 것은 Power 값을 증가시키면 됩니다.



<Blinn-Phong 반사: s0v\_07\_3blinn.zip>

#### 2.7.4 Lighting Assemble

지금까지 풍 반사와 풍 반사를 개선한 Blinn-Phong 반사를 살펴보았습니다. 게임에서는 Lambert 확산과 함께 이들을 결합해서 조명 효과를 만듭니다.



<Lambert+Phong: [s0v\\_07\\_4lambert+phong.zip](#),

Lambert+Blinn-Phong: [s0v\\_07\\_5lambert+blinn.zip](#)>

이들 조명 효과는 다음과 같은 공식으로 출력 레지스터 oD0의 값을 설정했습니다.

출력 색상(oD0) = Lambert \* 정점 색상 + Blinn 또는

출력 색상(oD0) = Lambert \* 정점 색상 \* Blinn

그런데 이 방법 대로 조명 효과를 만들고 텍스처를 적용하기 위해서 쉐이더 코드를 추가하고,

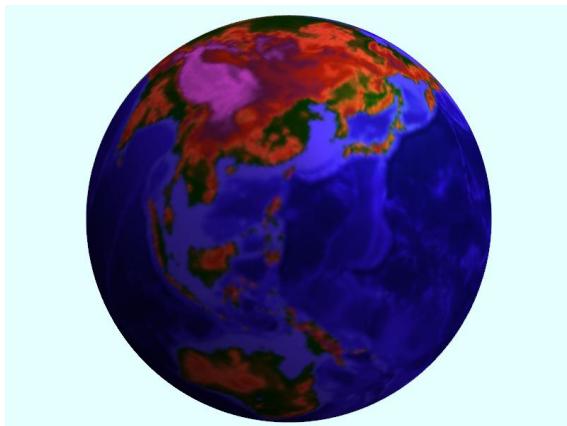
```
dcl_color      v2
dcl_texcoord   v3
```

```
...
```

```
mad oD0, Lmb, v2, Bln
```

```
mov oT0, v3
```

화면에 출력하면 조명 효과의 Specular 적용이 우리가 원하는 형태로 되고 있지 않음을 볼 수 있습니다.



<Specular 효과 적용이 미미한 예. [s0v\\_07\\_6texture1.zip](#)>

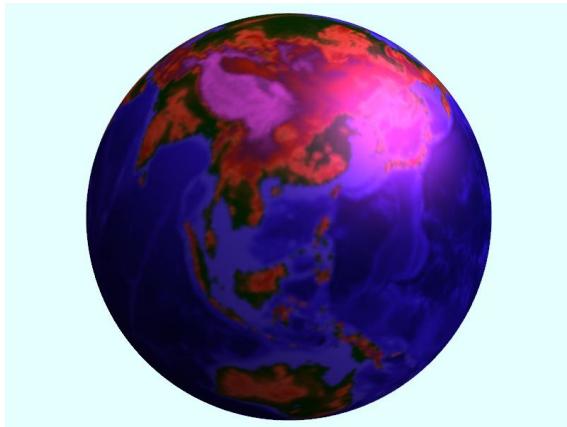
이것은 출력 레지스터 oD0가 Diffuse 값을 저장하는 용도로 사용되고, 고정 기능 파이프라인에서 픽셀 처리를 하게 되면 이 oD0의 색상 범위를 [0, 1]로 정규화 해서 사용하기 문입니다.

아직까지 우리는 픽셀 쉐이더를 사용하지 않기 때문에 고정 기능 파이프라인에서 이것을 개선하도록 합시다. 먼저 쉐이더 코드를 oD0에는 Lambert \* 정점 색상 결과를 출력하고 oD1는 Blinn 또는 Phong 반사의 Specular 값을 출력합니다.

```
mul oD0, Lmb, v2          // Output Diffuse: Lambert * Vertex Diffuse
mov oD1, Bln               // Output Specular: Blinn-Phong Reflectance
```

다중 텍스처 처리(Multi-Texturing)의 단계에서 색상 혼합 방법을 D3DTOP\_MULTIPLYADD로 정합니다. MULTIPLYADD는 Arg0 + Arg1 \* Arg2 연산을 수행하기 때문에 색상 인수 0번은 Specular(oD1)으로 설정하고 Arg1은 텍스처를 Arg2는 Diffuse(oD0)로 설정합니다.

```
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG0, D3DTA_SPECULAR);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MULTIPLYADD);
```



<Specular 효과. [s0v\\_07\\_6texture2.zip](#)>

## 2.8 Vertex Blending

정점 블렌딩(Vertex Blending)은 정점의 위치에 비중을 추가해서 최종 위치를 구하는 방법으로 대표적인 예가 스키닝(Skinning)입니다. 정점 블렌딩은 간단하게 다음과 같이 간단한 공식으로 표현합니다.

$$\vec{P} = \sum \vec{p}_i * W_i$$

대부분 정점에 행렬이 적용되어 이것을 일반적으로 표현하면  $\vec{P} = \sum \vec{p}_i * \vec{M}_i * W_i$  이 되며 모든

$\vec{p}_i$  가 같고  $1 = \sum W_i$  이면 스키닝이 됩니다.

$$\vec{P} = \sum \vec{p}_i * \vec{M}_i * W_i \rightarrow \vec{P} = \sum \vec{p} * \vec{M}_i * W_i = \vec{p} * \sum \vec{M}_i * W_i$$

정점 쉐이더를 이용해서 정점 블렌딩을 구현하기 위해서 3D 기초 과정에서 구현했던 태극기 예제의 폴리곤을 이용해 보겠습니다.

먼저 다음과 같이 2개의 비중(Weight)이 있는 정점 구조체를 선언합니다.

```
struct VtxBlend
{
    D3DXVECTOR3 p;           // 위치
```

```

FLOAT      g;          // 비중(Weight)
DWORD      d;          // Diffuse
FLOAT      u, v;       // 텍스처 좌표
enum { FVF = (D3DFVF_XYZB1 | D3DFVF_DIFFUSE | D3DFVF_TEX1), };
...
};


```

이 정점 구조체로 만들어진 정점 데이터의 비중을 정점 쉐이더에서 사용하기 위해서 입력 레지스터를 선언해야 합니다.

```
dcl_blendweight      v1      // 행렬 비중
```

하나의 정점에 두 개의 행렬이 적용된 변환에서 블렌딩 정점의 위치는 다음과 같이 계산 됩니다.

블렌딩 정점 위치 = 정점 위치 \* 행렬0 \* 비중 + 정점 위치 \* 행렬1 \* (1 - 비중)

이것을 쉐이더로 표현하면 외부에서 두 개의 월드 행렬을 적용해서 정점의 위치를 각각 저장한 후에 행렬0으로 변환된 위치에는 "비중(Weight)"을 곱하고 "1-비중"을 계산해서 행렬1으로 변환된 정점에 곱한 다음 둘을 더해서 정점 블렌딩을 구현합니다.

```

m4x4 r0, v0, c12           // 월드 행렬 0에 의한 정점 변환
m4x4 r1, v0, c16           // 월드 행렬 1에 의한 정점 변환
mul r0, r0, v1.x           // r0 = r0 * weight

add r2, c0.x, -v1.x         // r2.xyzw = 1 - weight
mad r2, r1, r2, r0          // pos = (1-weight)*v1 + v0*weight

```

계산을 빠르게 적용하기 위해 mad를 이용했습니다. 마지막으로 뷰 변환, 투영 변환을 적용하고 출력 레지스터에 복사합니다.

```

m4x4 r0, r2, c4             // 뷰 변환
m4x4 oPos, r0, c8            // 투영 변환

```



<정점 블렌딩: [s0v\\_08\\_vertex\\_blending.zip](#)>

대부분 스키닝은 4개의 행렬 인덱스를 포함한 정점으로 구현됩니다.

```
struct VtxBlend
{
    D3DXVECTOR3      p;
    FLOAT           g[3];        // 비중(Weight)
    BYTE            m[4];        // 행렬의 인덱스
    ...
    enum { FVF = (D3DFVF_XYZB4 | D3DFVF_LASTBETA_UBYTE4 | ...), };
};
```

정점 쉐이더는 비중과 인덱스를 처리하기 위한 입력 레지스터 선언이 필요합니다. 정점 데이터에서 인덱스를 가져와야 하는데 인덱스의 데이터 타입은 정수형입니다. 정수형 데이터를 저장하려면 "mova"와 같은 명령어를 이용해야 하고 이를 위해서 정점 쉐이더 버전은 2.0이상이 필요합니다. 또한 행렬은 4x4 float 형입니다. 따라서 인덱스에 4를 곱해야만 외부에서 스키닝에 적용할 행렬 배열을 상수 레지스터에 복사했을 때 이 상수 레지스터에 저장된 행렬 값을 제대로 가져올 수 있게 됩니다.

[vs\\_2\\_0](#)

```
def      c0, 4.0, 0.0, 0.0, 1.0

dcl_position      v0      // 위치
dcl_blendweight   v1      // 행렬 비중
dcl_blendindices  v2      // 행렬 인덱스
...
```

```
mul r0, v2, c0.x      // 상수 레지스터의 위치를 정하기 위해 인덱스에 4를 곱한다.
mov a0, r0             // 상수 레지스터 위치 저장
```

쉐이더의 레지스터는 [] 연산자를 이용해서 데이터를 가져올 수 있습니다. 상수 레지스터에 복사된 행렬 값을 []연산자로 가져와서 4개의 인덱스에 대한 변환을 구현 합니다.

```
#define MATRIX_OFFSET    12
...
m4x4 r0, v0, c[MATRIX_OFFSET + a0.x]
m4x4 r1, v0, c[MATRIX_OFFSET + a0.y]
m4x4 r2, v0, c[MATRIX_OFFSET + a0.z]
m4x4 r3, v0, c[MATRIX_OFFSET + a0.w]
```

모든 비중의 합은 1입니다. 그런데 정점 구조체의 비중은 "float g[3]"으로 되어 있어서 입력 레지스터는 3개의 비중만 전달될 것이므로 입력 레지스터에서 비중을 먼저 복사하고 마지막 비중을 "1- (v1.x + v1.y + v1.z)"으로 계산합니다.

```
def    c0, 4.0, 0.0, 0.0, 1.0
...
mov    r4, v1
add    r4.w, r4.x, r4.y
add    r4.w, r4.w, r4.z
add    r4.w, c0.w,-r4.w
```

변환된 정점의 위치에 각각 비중을 곱하고 이 위치들을 전부 더합니다.

```
// 변환된 정점의 위치에 각 비중을 곱한다.
mul r0, r0, r4.x
mul r1, r1, r4.y
mul r2, r2, r4.z
mul r3, r3, r4.w

// 비중이 곱해진 각 위치를 더한다.
add    r0, r0, r1
add    r0, r0, r2
add    r0, r0, r3
```

뷰 변환, 투영 변환 행렬을 적용하고 출력 레지스터에 복사하면 스키닝이 완성됩니다.

```
m4x4 r1, r0, c4      // 뷰 변환
m4x4 oPos, r1, c8    // 투영 변환
```



<정점 쉐이더 스키닝: [s0v\\_08\\_vertex\\_skinning.zip](#)>

## 2.9 Fog Effect

정점 쉐이더 중에서 흥미로운 부분이 포그(Fog)입니다. 포그는 정점의 위치를 가지고 결정합니다. 고정 기능 파이프라인에서의 포그는 카메라의 거리 또는 카메라의 z 축에 대한 값에 의존하기 때문에 높은 산에 올라가면 구름이 발 밑에 걸리는 높이 포그(Layered Fog) 같은 것들은 표현이 불가능 합니다.

쉐이더는 공식만 정해지면 포그에 대해서 아주 쉽게 구현할 수 있습니다. 예를 들어 고정 기능 파이프라인과 동일한 카메라의 Z 축 방향의 거리에 의존하는 선형 포그(Linear)의 계수는 다음과 같이 계산합니다.

$$\text{Fog Factor} = \text{뷰 변환 후 정점의 } z \text{ 값} / (\text{포그 끝 값} - \text{포그 시작 값})$$

'뷰 변환 후의 정점 z값'을 사용하는 이유는 뷰 행렬을 월드 변환한 정점에 곱하면 카메라 공간의 z축에 대한 값으로 계산 되기 때문입니다.

$$\text{뷰 변환 후 정점 } z = \text{dot}(\text{월드 변환 정점} - \text{카메라 위치}), \text{카메라 } z\text{축 벡터)$$

공식들이 간단해서 어렵지 않게 쉐이더를 작성할 수 있습니다.

```
dcl_position    v0
...
m4x4 r0, v0, c4      // 뷰 변환 후 정점 z
```

뷰 변환 후의 정점 z를 구했고 다음으로 Fog Factor를 구합니다.

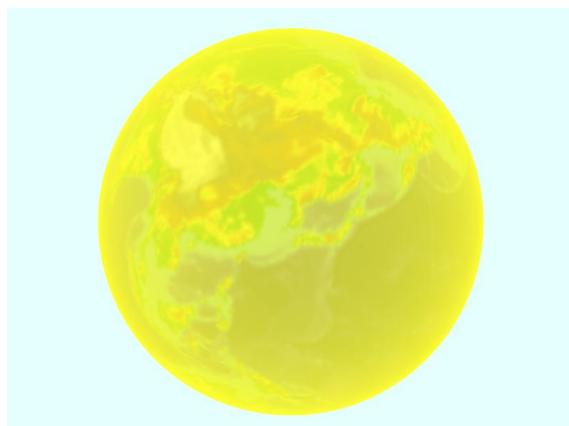
```
#define FogBgn  c12.x  // 포그 시작 값
#define FogEnd   c12.y  // 포그 끝 값
#define FogDsR   c12.w  // 1/(포그 끝 값 - 포그 시작 값)
...
sub r0.z, FogEnd, r0.z // (fog end - distance)
mul r0.x, r0.z, FogDsR // Fog Factor = distance/(end - begin)
```

Fog Factor가 [0, 1] 범위에 있도록 하기 위해서 min, max 를 이용합니다.

```
def c14, 1.0, 0.0, 0.0, 0.0
...
min r0.x, r0.x, c14.x // 1 보다 큰 값 제거
max r0.x, r0.x, c14.y // 음수 값 제거
```

마지막으로 Fog Factor를 출력 레지스터 oFog에 복사 합니다.

```
mov oFog, r0.x
```



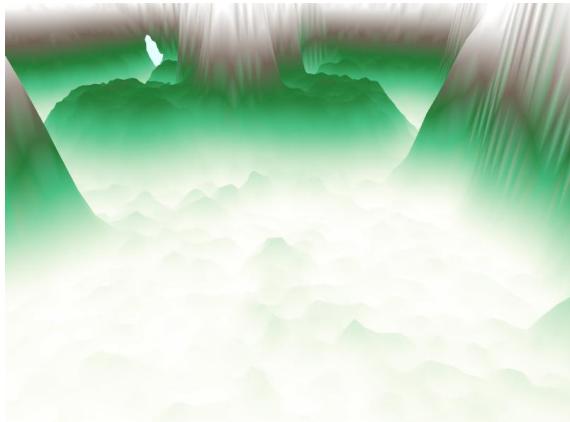
<선형 포그(Linear Fog): [s0v\\_09\\_fog1\\_range.zip](#)>

고정 기능 파이프라인의 Range Fog는 뷰 변환 후의 정점 z에 의존하는 포그입니다. 높이 포그 (Layered Fog)는 월드 변환 후의 정점 y에 대한 포그이기 때문에 Fog Factor 공식은 아주 간단합니다.

**Layered Fog Factor** = 월드 변환 후 정점의 y 값/(포그 끝 값 - 포그 시작 값)

월드 변환이 없는 정점의 경우에는 Fog Factor를 쉐이더로 작성하는 것은 어려운 일이 아닙니다.

```
#define FogBgn c12.x
#define FogEnd c12.y
#define FogDsR c12.w // 1/(FogEnd-FogBgn)
...
def c14, 1.0, 0.0, 0.0, 0.0
dcl_position v0
...
mul r0.x, v0.y, FogDsR // Output FogFactor = height/(end - begin)
min r0.x, r0.x, c14.x // 1 보다 큰 값 제거
max r0.x, r0.x, c14.y // 음수 값 제거
mov oFog, r0.x // 포그 출력 레지스터에 저장
```



<높이> 포그(Layered Fog): [s0v\\_09\\_fog2\\_height.zip](#)

지금까지 Fog Factor를 계산하고 출력 레지스터 oFog에 복사했습니다. 이런 방식은 렌더링에서 디바이스의 포그를 활성화(D3DRS\_FOGENABLE, TRUE) 해야 합니다. 또한 높이 포그는 D3DRS\_FOGTABLEMODE 를 D3DFOG\_NONE으로 설정해야 합니다.

쉐이더 버전 3.0 이상에서는 oFog를 사용할 수 없어서 포그를 직접 구현 해야 합니다. DXSDK의 도

움말을 보면 포그가 적용될 때 고정 기능 파이프라인에서 정점의 최종 색상은 포그 색상, Fog Factor, 조명과 정점의 색상 혼합으로 만들어진 Diffuse 값을 선형 보간 형식으로 결정됩니다.

정점의 최종 색상 = Fog 색상 \* Fog Factor + Diffuse \* (1 - Fog Factor)

높이 포그의 예제를 수정해서 쉐이더를 적용해 봅시다. Fog Factor 계산을 mad 연산자로 한 번에 처리하기 위해서 공식을 풀어줍니다.

$$\begin{aligned}\text{Fog Factor} &= (\text{포그 끝 값} - \text{월드 변환 후 정점의 } y \text{ 값}) / (\text{포그 끝 값} - \text{포그 시작 값}) \\ &= (\text{포그 끝 값}) / (\text{포그 끝 값} - \text{포그 시작 값}) \\ &\quad - \text{월드 변환 후 정점의 } y \text{ 값} / (\text{포그 끝 값} - \text{포그 시작 값})\end{aligned}$$

$$\begin{aligned}\text{Fog Factor} &= -\text{월드 변환 후 정점의 } y \text{ 값} / (\text{포그 끝 값} - \text{포그 시작 값}) \\ &\quad + \text{포그 끝 값} / (\text{포그 끝 값} - \text{포그 시작 값})\end{aligned}$$

외부에서 포그 끝 값 / (포그 끝 값 - 포그 시작 값)를 계산한다고 가정하고 다음과 같은 매크로를 정의 합니다.

```
#define FogFct c12.z // FogEnd/(FogEnd - Begin)
#define FogDsR c12.w // 1.0/(FogEnd-FogBgn)
```

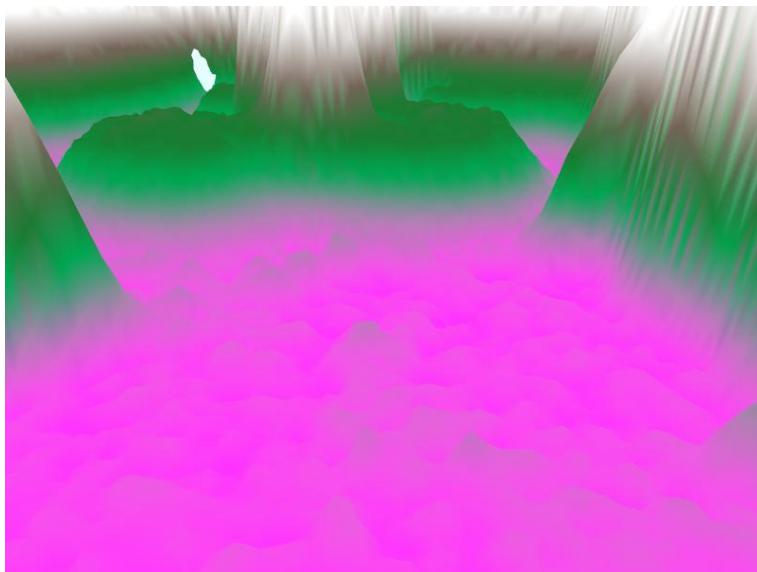
이렇게 정의된 매크로가 있으면 y 값에 의존하는 높이 포그의 Fog Factor r0.x는 mad로 한 번에 계산 됩니다.

```
mad r0.x, -v0.y, FogDsR, FogFct
```

min, max 연산자로 [0, 1] 범위로 만들고 정점의 최종 색상을 선형 보간 형식으로 만들고 출력 레지스터 oD0에 복사 합니다.

```
mul r1, FogColor, r0.x
add r2, c14.x, -r0.x // (1-r0.x) <== (1-w)
mad oD0, v2, r2, r1 // Diffuse *(1-FogFactor) + FogColor * FogFactor
```

이렇게 되면 고정 기능 파이프라인의 어떤 설정도 필요 없이 순수한 쉐이더로 포그를 구현할 수 있습니다.



<완전한 쉐이더로 구현된 높이 포그: [s0v\\_09\\_fog3\\_shader.zip](#)>

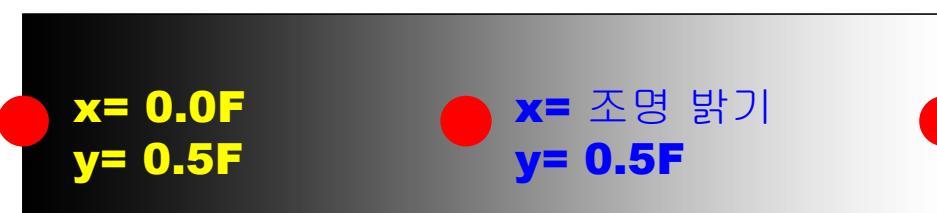
## 2.10 Toon Shading

정점 쉐이더의 응용 중에 하나가 툰 쉐이딩(Toon: cartoon Shading)입니다. 툰 쉐이딩을 간단하게 설명하면 정점에 의한 반사 밝기를 선형적으로 처리하지 않고 양자화 단위의 단계적 처리를 의미합니다.

고정파이프 라인에서 만약 정점 a의 라이팅에 대한 반사의 세기가 0.3이고 정점 b의 라이팅에 세기가 0.7이면 중간 밝기는 선형적인 계산을 통해 보간합니다. 하지만 툤 효과는 반사의 밝기를 선형적으로 처리하지 않고 마치 계단처럼 특정한 범위 내에서는 같은 밝기로 처리합니다.

만약 고정 기능 파이프라인에서 툤 효과를 만들기 위해서는 렌더링의 여러 패스를 거쳐 가야 하지만 정점 쉐이더를 사용하면 아주 간단하고 쉽게 처리할 수 있습니다.

정점의 조명 효과에 대한 밝기를 oD0에 출력했습니다. 그런데 밝기의 세기는 [0, 1] 범위이며 이것을 oD0에 출력하지 않고 텍스처 좌표로 출력하면 굀셀 처리 과정에서 다음 그림과 같이 연속적으로 변하는 텍스처에서 색상을 샘플링 하게 되면 샘플링 된 색상은 곧, 조명의 밝기와 동등한 결과가 됩니다.



<밝기가 0, [0, 1] 범위, 1에서의 샘플링 위치>

쉐이더는 값을 정하지 않으면 0 또는 1이 되기 때문에 조명의 밝기를 텍스처에서 가져올 때 쉐이더에서 y는 대부분 설정을 안 합니다.

Lambert 확산에 대해서 밝기의 계산은 dot(정점 법선 벡터, 빛의 방향 벡터)입니다.

```
#define Lgt      -c8
...
def c24, 1.0, 0.5, 0.1, .9

dcl_normal      v1           // 정점 법선
...
m3x3 r0, v1, c4          // 법선 벡터에 대한 변환
dp3  r1, r0, Lgt          // 밝기 = Dot(변환된 법선 벡터, 빛의 방향 벡터)

mad r1.x, r1.x, c24.y    // (Dot + 1)* 0.5 = Dot * 0.5 + 0.5
```

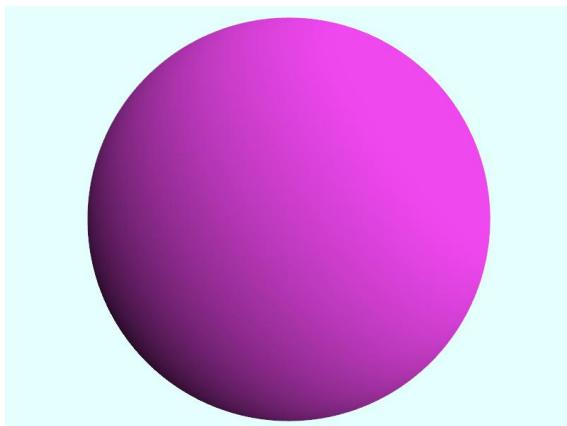
Lambert 확산에 대한 반사 세기는 "dot(정점 법선 벡터, 조명 방향 벡터)" 가 되어 전체 크기는  $\cos \theta$ 에 비례하고 값의 범위는 [-1, 1]이 됩니다. 이 값의 범위를 [0, 1]으로 하기 위해서 "mad" 연산자와 0.5 값을 이용했습니다.

계산된 반사의 밝기를 출력 레지스터 oD0에 복사하는 대신 oT0.x에 저장합니다.

```
mov oT0.x, r1.x           // 결과를 텍스처 좌표 x에 저장.
```

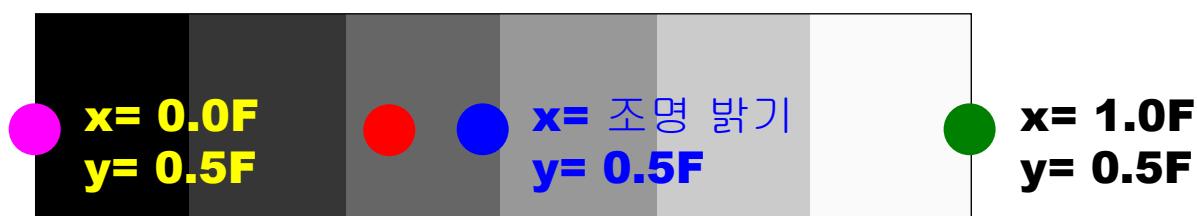
픽셀 처리를 고정 파이프라인을 이용하려면 다중 텍스처를 Texture와 Diffuse의 혼합으로 설정하고 폴리곤을 렌더링 합니다.

```
m_pDev->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE);
m_pDev->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_DIFFUSE);
m_pDev->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE);
```



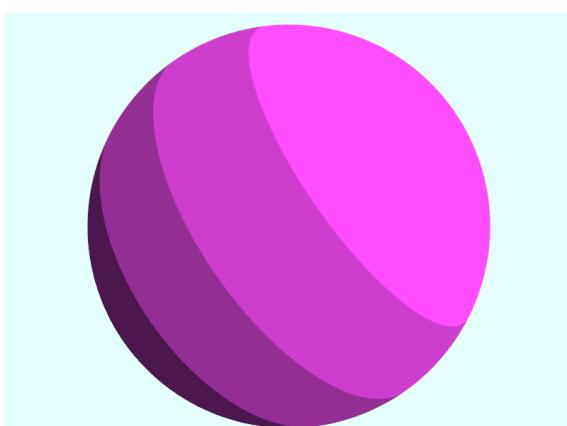
<Lambert 확산을 텍스처 좌표로 사용. [s0v\\_10\\_toon1\\_texture\\_lighting.zip](#)>

툰 쉐이딩은 연속적으로 변화하는 텍스처 대신 다음과 같이 밝기가 양자화된 텍스처를 사용합니다.



그림에서 조명의 밝기 차이로 인해 파란색 점과 붉은색 점의 텍스처 좌표의 위치는 다르지만 텍스처에서 샘플링 하는 픽셀의 색상은 동등합니다.

이전의 Lambert 확산에 대해서 연속적으로 변하는 텍스처 대신 양자화된 텍스처를 적용하면 다음 그림과 같이 툴 쉐이딩(Toon Shading)이 적용된 장면을 볼 수 있습니다.



<툰 쉐이딩(Toon Shading): [s0v\\_10\\_toon1\\_texture\\_toon.zip](#)>

툰 쉐이딩에서 사용하는 텍스처는 x만 사용하기 때문에 높이가 필요 없습니다. 1차원 텍스처는 높이가 하나의 픽셀로 구성하면 됩니다. 1차원 텍스처 만드는 것은 간단해서 그래픽 툴을 이용하는

것 보다 프로그램에서 실시간으로 만드는 것이 정보 보호를 위해서 이점이 있습니다.

D3DXCreateTexture() 함수를 사용하면 실시간으로 텍스처를 만들 수 있으며 텍스처 객체의 LockRect() 함수를 사용해서 픽셀 데이터를 가져와서 수정 할 수 있습니다.

```

hr = D3DXCreateTexture(m_pDev , 512, 1 , 0, 0
                      , D3DFMT_X8R8G8B8, D3DPOOL_MANAGED, &m_pTex );
if ( FAILED(hr) )
    return hr;

D3DLOCKED_RECT pRect;
m_pTex->LockRect(0, &pRect, NULL, 0);

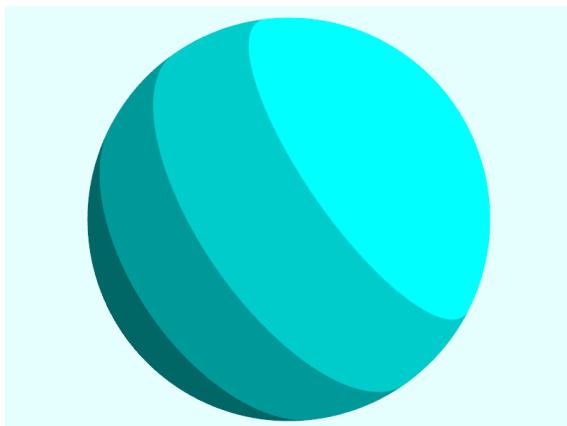
DWORD* pColor = (DWORD*)pRect.pBits;
for( INT i = 0 ; i < 512; ++i)
{
    FLOAT c = 0;

    if(i<10)      c = 0;
    else if(i<100) c= 0.2f;
    else if(i<200) c= 0.4f;
    else if(i<300) c= 0.6f;
    else if(i<400) c= 0.8f;
    else           c= 1.0;

    pColor[i] = D3DXCOLOR(c,c,c,1);
}

m_pTex->UnlockRect(0);

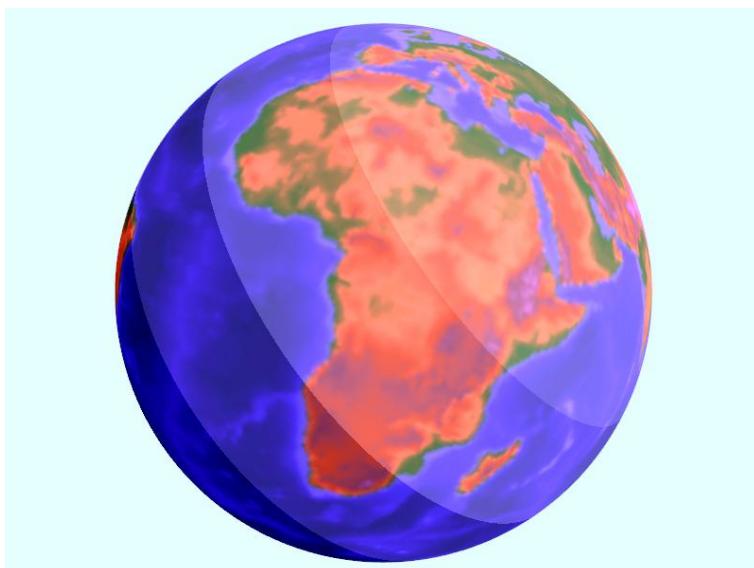
```



<1차원 툰 쉐이딩 텍스처: [s0v\\_10\\_toon2\\_1D\\_texture.zip](#)>

툰 쉐이더용 텍스처를 사용하지 않고 직접 조건 문을 사용해서 툰 쉐이딩을 구현하는 방법도 있습니다. 그런데 저 수준으로 약간 난이도 있는 조건 문을 작성하는 것보다 고 수준 언어의 조건 문이 훨씬 간단 하므로 이후 HLSL에서 텍스처 없이 툰 쉐이딩을 구현해 보도록 하겠습니다.

조명을 툰 쉐이딩으로 처리하고 Diffuse용 텍스처와 다중 텍스처 처리로 혼합을 하게 되면 툰 쉐이딩의 기본적인 내용은 마무리가 됩니다.



<툰 쉐이딩 + Diffuse Map: [s0v\\_10\\_toon2\\_diffuse+toon.zip](#)>

[s0v\\_10\\_toon2\\_diffuse+toon.zip](#)의 멀티 텍스처 처리 방식은

$$\text{최종 색상} = \text{Diffuse Map} * \text{Diffuse Color} + \text{Toon Shading} - 0.5$$

이를 고정 기능 파이프라인에서 다중 텍스처 처리를 다음과 같이 작성했습니다.

```

m_pDev->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE);
m_pDev->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_DIFFUSE);
m_pDev->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE);

m_pDev->SetTextureStageState( 1 , D3DTSS_COLORARG1 , D3DTA_CURRENT);
m_pDev->SetTextureStageState( 1 , D3DTSS_COLORARG2 , D3DTA_TEXTURE);
m_pDev->SetTextureStageState( 1 , D3DTSS_COLOROP , D3DTOP_ADDSIGNED);
...
m_pDev->SetTexture( 0, m_pTxDif );
m_pDev->SetTexture( 1, m_pTxToon );

```

## 2.11 윤곽선(Edge)

윤곽선을 만드는 방법은 변환한 정점의 깊이 값, 정점의 ID, 법선 벡터 등을 이용해서 만드는데 간단하게 윤곽선을 만들고자 할 때는 정점의 법선 벡터를 이용하는 것이 편리합니다.

모델 좌표계에서 윤곽선에 해당하는 정점 위치는 다음과 같이 만들 수 있습니다.

윤곽선 정점 위치 = 정점 위치 + 법선 벡터 \* 크기

렌더링은 2번 진행 합니다. 먼저 일반적인 정점을 렌더링하고 다음으로 윤곽선 정점의 위치를 렌더링 합니다.

고정 기능 파이프라인에서는 윤곽선 정점 위치에 대해서 정점 버퍼를 새로 만들어야 하지만 쉐이더를 사용하면 기존에 있는 정점 버퍼를 그대로 사용하고 대신 쉐이더에서 정점의 법선 벡터와 크기 값을 이용해서 윤곽선 정점 위치를 설정합니다.

```

#define Scl c27
...
def c25, 0.0, 0.0, 0.0, 1.0

dcl_position    v0      // 정점 위치 벡터 레지스터 선언 v0
dcl_normal      v1      // 정점 법선 벡터 레지스터 선언 v1

mov r0, v1          // 법선 벡터
mad r0, r0, Scl, v0 // 윤곽선 위치' = 법선 벡터 * 스케일 + 위치

```

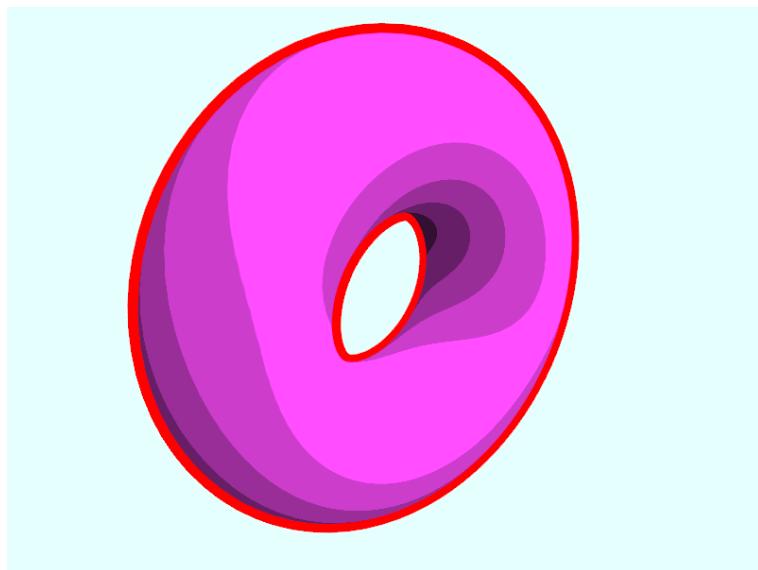
```
mov r0.w, c25.w      // 윤곽선 위치' w = 1.0
m4x4 oPos, r0, c0    // 변환
```

같은 정점 베파를 가지고 두 번 그리는데 첫 번째에서는 CCW로 렌더링 합니다.

```
// Toon Shading Process
m_pDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
...
m_pDev->SetTexture( 0, m_pTxToon );
m_pDev->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, m_nVtx, 0, m_nFce );
```

두 번째는 CW로 그립니다. 이렇게 하면 CCW로 그린 것이 앞쪽에 나오게 됩니다.

```
// Edge Process
m_pDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
...
m_pDev->SetTexture( 0, NULL );
m_pDev->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, m_nVtx, 0, m_nFce );
```



<윤곽선: 위치 + 법선 벡터\* 크기. [s0v\\_10\\_toon3\\_edge1.zip](#)>

정확한 윤곽선 대신 대충 만드는 윤곽선도 있습니다. 이 윤곽선은 카메라의 (-)z축과 법선 벡터의 내적 결과를 툰 쉐이딩과 같이 특정 텍스처의 좌표로 설정하는 것입니다.

윤곽선 텍스처 좌표 = dot(변환된 법선 벡터, 카메라 (-)z축)



### <윤곽선 텍스처>

지금까지 내용을 가지고 DX Tiny의 모델에 대해서 Diffuse Map, 툰 쉐이딩, 윤곽선 텍스처 적용을 쉐이더로 작성해 봅시다.

쉐이더 프로그램 작성은 편리하게 하기 위해서 다음과 같이 매크로를 정의합니다.

```
#define Nor      r2
#define Lgt      -c8
#define CamZ     -c16
```

입력 레지스터는 위치, 법선 벡터, Diffuse, 텍스처 좌표들로 선언하고 지정합니다.

```
dcl_position    v0      // 정점 위치 레지스터 선언 v0
dcl_normal      v1      // 정점 법선 레지스터 선언 v1
dcl_color0      v2      // 정점 디퓨즈 색상
dcl_texcoord0   v3      // 정점 텍스처 좌표
```

먼저 정점 위치 변환과 정점의 Diffuse 값, Diffuse Map 텍스처 좌표를 출력 레지스터에 복사합니다.

```
m4x4 oPos, v0, c0      // 출력 위치
mov oD0, v2              // 정점 색상
mov oT0, v3              // Diffuse Map 텍스처 좌표
```

툰 쉐이딩에 대한 텍스처 좌표를 구현하고 oT1에 복사합니다.

```
def c24, 1.0, 0.5, 0.1, 0.9
...
m3x3 Nor, v1, c4        // 법선 벡터에 대한 변환
dp3 r1, Nor, Lgt         // 정점 밝기 계산
mad r1.x, r1.x, c24.y, c24.y
mov oT1.x, r1.x          // 결과를 텍스처 좌표 x에 저장
```

윤곽선에 대한 텍스처 좌표를 구하고 이 결과를 oT2에 복사합니다. 텍스처 좌표는 카메라의 -

$z$  축과 변환된 정점의 법선 벡터의 내적으로 계산됩니다.

```
dp3 r3.x, Nor, CamZ      // Edge = dot(N, (-)CameraZ)
mov oT2.x, r3.x
```

고정 기능 파이프라인에서 다중 텍스처 처리 연산은 "정점 Diffuse" \* "Diffuse Map" \* "Toon Shading" \* "윤곽선"으로 계산합니다. 이를 구현하면 다음과 같습니다.

```
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);

m_pDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT);
m_pDev->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);

m_pDev->SetTextureStageState(2, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(2, D3DTSS_COLORARG2, D3DTA_CURRENT);
m_pDev->SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_MODULATE);
```



<Diffuse \* Toon \* Edge. [s0v\\_10\\_toon3\\_edge2.zip](#)>

## 2.12 Depth Encoding

백 버퍼에 저장되는 깊이 값은 때로는 블롭 포그나 투영 그림자 처리에서 종종 이용되기도 합니다. 정점이 그래픽 파이프라인의 변환을 거치면 Z축의 값은 [0, 1]의 값을 가지게 되어서 이것을 색상으로 사용해도 되지만 변환 후에 z는 실제로 1.0 근처에 몰려 있습니다. 따라서 적당한 값을 정규 변환 후에 z값에 곱하고 이 것을 색상으로 사용하는 것이 좋습니다.

정점의 변환 값을 색상으로 사용하기 때문에 임시 레지스터에 월드, 뷰, 투영 행렬의 곱으로 구성된 행렬에 정점의 위치를 곱한 값을 저장합니다.

```
dcl_position    v0          // 정점 위치를 입력 레지스터 v0에 선언
m4x4 r0, v0, c0          // 정점의 변환: 입력 위치 * c0에 입력된 행렬
```

다음으로 출력 레지스터 oPos에 임시 레지스터에 저장된 변환된 정점 위치를 복사합니다.

```
mov oPos, r0          // 출력 위치 = r0, z=[0,1]
```

Depth Range 값을 저장한 상수 레지스터의 값과 임시 레지스터에 저장된 변환된 정점 위치의 z값을 곱하고 이것을 출력 레지스터 oDO에 복사합니다.

```
#define DepthRange      c26
...
mul oDO, r0.z, DepthRange.z    // 출력 디퓨즈 색상 = 변환 후 정점의 z * DepthRange.z
```



<깊이 값 렌더링. [s0v\\_11\\_depth.zip](#)>

## 2.13 Vertex Shader Effect

쉐이더에서 행렬을 정점 쉐이더의 상수 레지스터에 설정할 때 매번 전치(Transpose) 하는 것이 종

종 개발자를 혼란하게 만들거나 때로는 전치를 하지 않고 연결하는 실수를 종종 합니다. 또한 SetVertexShaderConstantF() 함수에서 Float 형, Vector 형, Matrix 형 등의 다른 타입을 하나의 함수에 설정하다 보니 float4형의 개수를 인수로 전달해야 합니다.

이것은 D3D 디바이스가 저 수준을 지원하기 때문에 인터페이스를 늘리지 않기 위해서 어쩔 수 없이 만들어진 형태입니다.

만약 다음과 같이 저 수준 정점 쉐이더의 인터페이스를 만든다면 쉐이더 사용을 좀 더 편리하게 사용할 수 있습니다.

```
interface ILcShader
{
    virtual INT      Begin()=0;
    virtual INT      End()=0;

    virtual INT      SetFVF(void* pFVF)=0;
    virtual INT      SetMatrix(INT nRegister, const D3DXMATRIX* v, INT Count=1)=0;
    virtual INT      SetVector(INT nRegister, const D3DXVECTOR4* v)=0;
    virtual INT      SetColor(INT nRegister, const D3DXCOLOR* v)=0;
    virtual INT      SetFloat(INT nRegister, const FLOAT* v)=0;
};
```

[s0v\\_12\\_ShaderEffect.zip](#)은 ILcShader와 ILcShader를 상속 받은 CLcShader 클래스가 구현되어 있습니다.

Begin()/End() 함수는 정점 처리를 프로그램 가능한 파이프라인에서 처리를 하거나 해제하는 함수로 다음과 같이 구현되어 있습니다.

```
INT CLcShader::Begin()
{
    return m_pDev->SetVertexShader(m_pShd);
}
```

```
INT CLcShader::End()
{
    m_pDev->SetVertexDeclaration(NULL);
    return m_pDev->SetVertexShader(NULL);
}
```

SetFVF()는 고정 기능 파이프라인의 FVF() 함수 호출과 유사한 기능을 하기 때문에 이 이름이 붙었습니다.

```
INT CLcShader::SetFVF(void* pFVF)
{
    return m_pDev->SetVertexDeclaration((PDVD)pFVF);
}
```

D3D 디바이스의 SetVertexConstantF() 함수를 분리해서 Matrix, Vector, Color, Float 형에 대해서 처리하도록 각 기능에 대한 명세를 분명히 하는 것이 프로그램 응집성에 도움이 됩니다.

```
INT CLcShader::SetMatrix(INT uReg, const D3DXMATRIX* v, INT nCount)
{
    HRESULT hr;
    for(int i=0; i<nCount; ++i)
    {
        D3DXMATRIX t;
        D3DXMatrixTranspose(&t, &v[i]);
        hr = m_pDev->SetVertexShaderConstantF( uReg + i*4, (FLOAT*)&t, 4);
        if(FAILED(hr))
            return -1;
    }
    return 0;
}
```

```
INT CLcShader::SetVector(INT uReg, const D3DXVECTOR4* v)
{
    return m_pDev->SetVertexShaderConstantF( uReg , (FLOAT*)v , 1);
}
```

```
INT CLcShader::SetColor(INT uReg, const D3DXCOLOR* v)
{
    return m_pDev->SetVertexShaderConstantF( uReg , (FLOAT*)v , 1);
}
```

```
INT CLcShader::SetFloat(INT uReg, const FLOAT* v)
{
```

```

    return m_pDev->SetVertexShaderConstantF( uReg , (FLOAT*)v , 1);
}

```

이렇게 구성된 ILcShader 인터페이스는 다음과 같이 간단하게 객체를 생성하고 사용할 수 있습니다.

```

ILcShader*      m_pVs;

// 함수를 통한 객체 생성
LcDev_CreateVertexShaderFromFile(&m_pVs, m_pDev, "data/Shader.vsh");

// 쉐이더 사용
m_pVs->Begin();
m_pVs->SetFVF(m_pFVF);

D3DXVECTOR4 DepthScalers(1.0f, 0.004f, 0.0f, 1.0f);
// 상수 연결
m_pVs->SetMatrix( 0, &(m_mtWld * mtViw * mtPrj));
m_pVs->SetVector(26, &DepthRange);
...
m_pDev->DrawPrimitive(...);

// 쉐이더 해제
m_pVs->End();

```

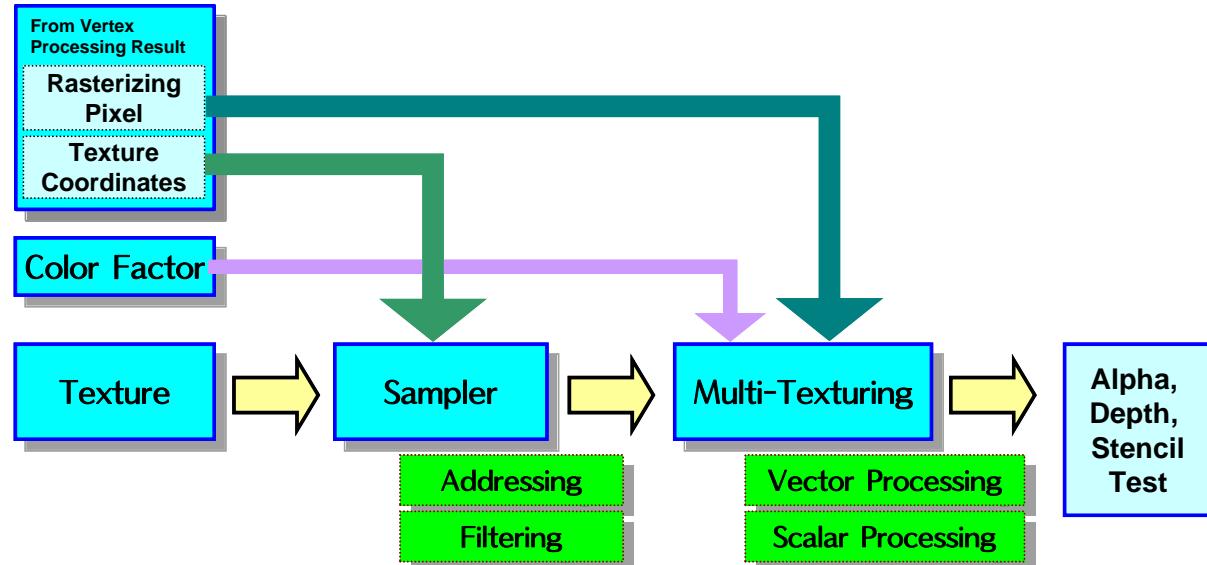
전체 코드는 [s0v\\_12\\_ShaderEffect.zip](#)를 참고하기 바랍니다.

### 3 Pixel Shader

지금까지 우리는 정점 셰이더 사용법을 살펴보았습니다. 정점 셰이더는 변환, 조명, 안개 효과 등 레스터라이징 이전까지의 처리 과정을 프로그램 가능한 파이프라인을 이용하는 것입니다.

Rasterizing 이후 만들어진 픽셀 데이터는 픽셀 처리 과정(Pixel Processing)으로 넘어갑니다. 픽셀 처리 과정은 샘플링→다중 텍스처 처리→알파 테스트→깊이 테스트→스텐실 테스트→픽셀 포그→알파 블렌딩 순으로 진행되고 마지막에 후면 버퍼를 갱신하는 과정입니다.

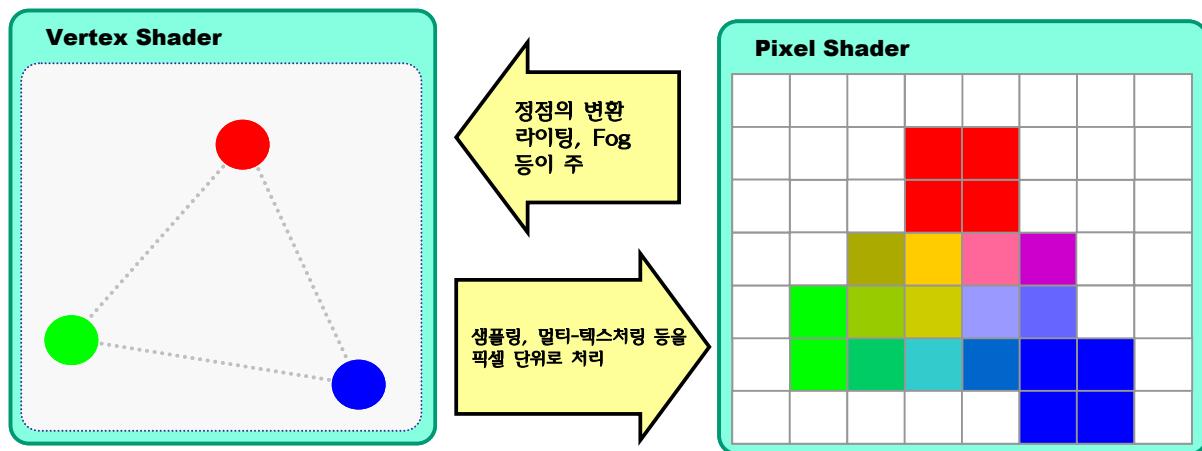
픽셀 셰이더는 이러한 픽셀 처리 과정 중에서 텍스처에서 색상을 추출하는 샘플링(Sampling)과 입력된 색상을 혼합하는 다중 텍스처 처리(Multi-Texturing)를 고정 기능 파이프라인이 아닌 프로그램 가능한 파이프라인으로 처리하는 것입니다.



<픽셀 처리 과정>

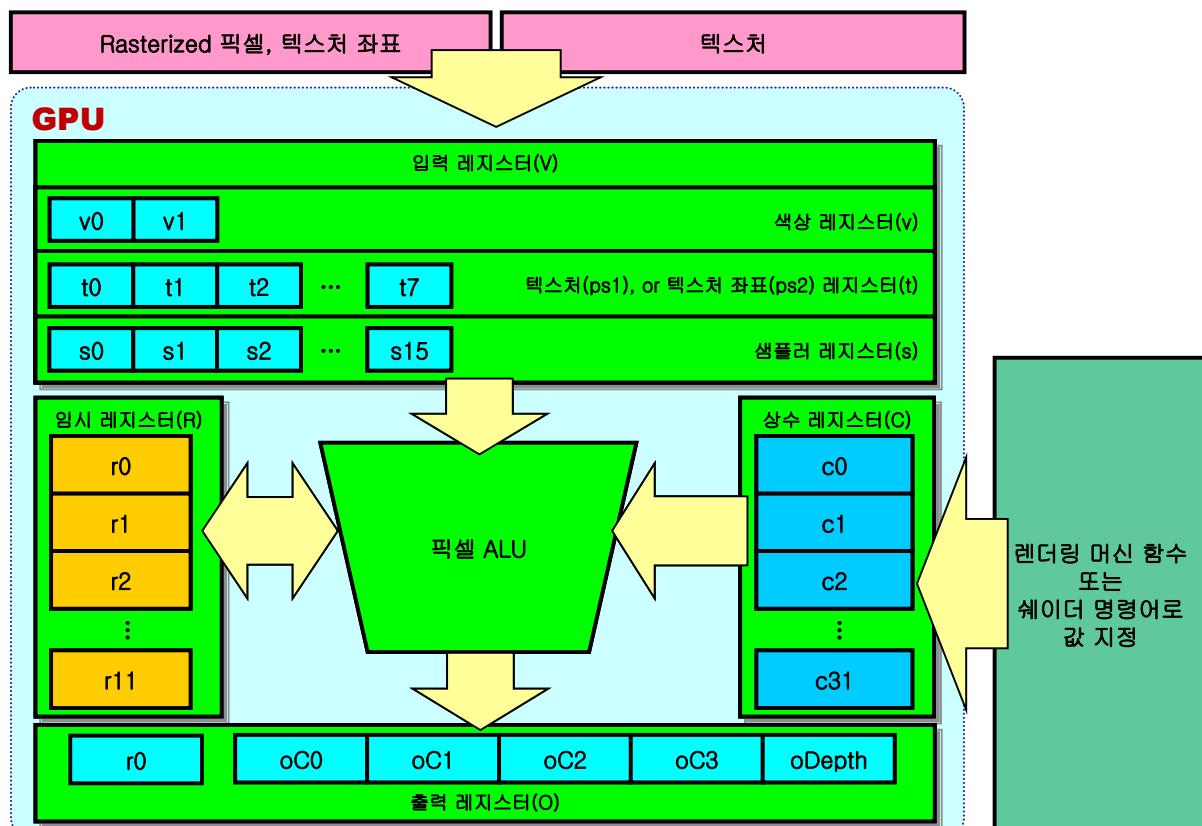
하나의 삼각형을 구성하기 위해서 세 개의 정점이 필요하지만 이 세 정점으로 구성된 삼각형의 픽셀을 채우는 작업은 화면 해상도와 카메라의 위치에 따라 달라집니다. 즉, 정점 처리보다 픽셀 처리가 훨씬 많은 작업이 필요합니다.

이와 같은 기술적 문제들과 픽셀 처리에 필요한 하드웨어의 구성에 대한 비용 문제로 인해서 픽셀 셰이더보다 정점 셰이더를 지원하는 GPU가 먼저 출시되었습니다. 또한 초기 GPU 제조사들은 저마다 자신들의 저 수준 픽셀 셰이더 언어를 표준으로 정하려고 했기 때문에 정점 셰이더와는 다르게 명령어들이 일관성이 부족했습니다.



&lt;정점 쉐이더와 퍽셀 쉐이더 역할 비교&gt;

기술 보다 이론이 앞서 있는 상황에서 아직까지 1.x 버전에서 작성한 많은 예제들이 있지만 퍽셀 쉐이더는 버전 2.0에 와서 많은 명령어들이 정리되었기 때문에 저수준으로 작성하면 각 버전마다 다른 명령어를 사용해야 하지만 고 수준으로 작성하면 거의 같은 함수를 사용할 수 있어서 여러분은 최소한 2.0이상 버전에서 고 수준으로 작성하는 것이 사용의 편리와 유지 보수를 위해서 좋습니다.



&lt;퍽셀 쉐이더 가상머신&gt;

픽셀 쉐이더 가상 머신은 정점 쉐이더 가상 머신처럼 입력 레지스터, 출력 레지스터, 상수 레지스터, 임시 레지스터, 그리고 산술과 논리 연산을 담당하는 ALU로 구성되어 있습니다.

입력 레지스터는 Rasterizing 을 거친 픽셀 데이터에 대해서 v로 시작을 합니다. 색상 레지스터 (Color Register) v#는 v0와 v1이 있고, v0는 Diffuse, v1은 Specular에 해당합니다.

t으로 시작하는 입력 레지스터 t#은 픽셀 쉐이더 버전마다 차이가 있습니다. 1.x 버전에서는 텍스처 좌표에 의해 샘플링 된 픽셀이지만 2.0 이후에는 텍스처 좌표 자체입니다.

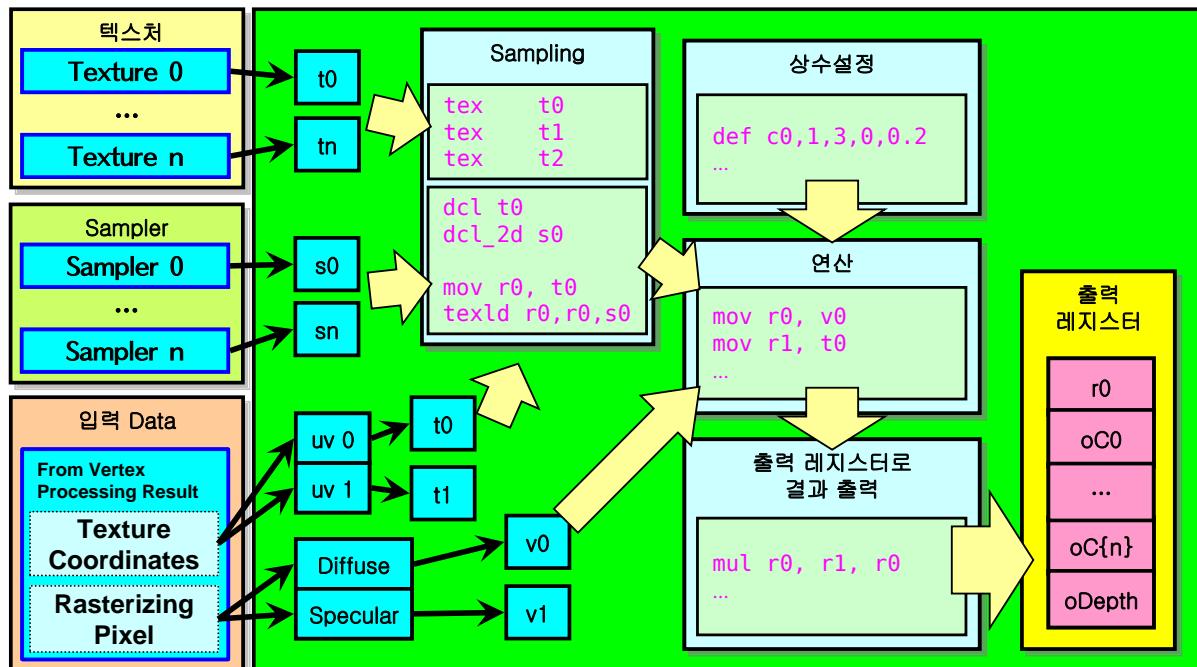
s로 시작하는 입력 레지스터 s#은 색상을 추출하는 샘플러(Sampler) 객체입니다.

임시 레지스터 r#은 정점 쉐이더의 임시 레지스터와 같은 기능을 수행하며 연산의 결과를 저장하는 용도로 사용되는 읽기, 쓰기 레지스터입니다.

상수 레지스터 c# 역시 정점 쉐이더의 상수 레지스터와 같은 기능을 수행하고 쉐이더 내부에서는 읽기만 가능하고 값의 설정은 외부에서 디바이스의 함수를 사용하거나 "def" 명령어로 미리 정해야 합니다.

출력 레지스터는 쉐이더 버전마다 크게 차이가 있습니다. 1.x 버전에서는 임시 레지스터 r0가 출력 레지스터입니다. 2.x 버전에서는 Multi-element texture가 가능해서 oC0~oC3이 있고, oDepth도 있습니다. 보통 oC0로 출력하는데 oC0는 후면 버퍼에 해당합니다.

픽셀 ALU는 픽셀 데이터에 대한 산술 연산과 논리 동작을 담당하며 기본적인 사칙 연산부터 픽셀에 대해서 내적, 제곱근, 승수, exp, log 등의 수학 함수들과 조건 문 등을 처리합니다.



<픽셀 쉐이더 레이아웃>

저 수준 픽셀 쉐이더 코드와 입력된 텍스처, 래스터 과정에 의해 만들어진 픽셀의 관계는 그림처

럼 먼저 입력 레지스터 선언에 따라 픽셀은 v#, 텍스처 좌표 또는 텍스처는 t#, 샘플러 객체는 #s 입력 레지스터에 저장됩니다. 이 입력 레지스터에 저장된 값과 미리 설정된 상수 레지스터 c#에 저장된 값을 가지고 픽셀 ALU는 연산을 합니다. 또한 명령어에 따라 연산의 결과를 임시 레지스터를 이용하고, 마지막으로 출력 레지스터에 복사 합니다.

100번 보는 것보다 한 번 작성해 것이 훨씬 이해가 빠르기 때문에 픽셀 쉐이더 연습을 통해서 하나하나 배워봅시다.

## 3.1 Simple Pixel Shader

### 3.1.1 Diffuse 출력

저 수준 픽셀 쉐이더의 명령어들과 문법은 정점 쉐이더와 거의 비슷합니다. 따라서 픽셀 쉐이더 코드를 컴파일 하고, 쉐이더 객체를 생성하고, 렌더링에서 상수 설정 등에 대한 기본 동작은 정점 쉐이더와 부분적으로 함수 이름만 다를 뿐 거의 같은 형식으로 되어 있습니다.

그런데 픽셀 쉐이더 1.x에서 각 버전마다 명령어들의 차이가 정점 쉐이더 보다 훨씬 심합니다. 예를 들어 ATI에서 ps\_1\_4 버전이 지원이 되지만 NVIDIA 계열에서는 ps\_1\_4를 지원 안 합니다. 따라서 여러분은 간단한 픽셀 쉐이더의 경우 1.1 버전을 사용하거나 아니면 2.0 이상 버전으로 사용하는 것이 좋습니다.

여기서는 픽셀 쉐이더 버전 1.1과 2.0을 중심으로 강의를 하겠습니다.

간단하게 색상이 있는 사각형을 출력하는 예를 만들어 봅시다. 이를 위해 다음과 같은 정점 구조체를 사용해야 합니다.

```
struct VtxD
{
    D3DXVECTOR3    p;        // 정점 위치
    DWORD         d;        // 정점 색상
    ...
    enum {FVF = (D3DFVF_XYZ|D3DFVF_DIFFUSE), };
};
```

이 구조체로 사각형을 출력하도록 4개의 정점을 만들고 위치와 색상을 설정합니다.

```
VtxD    m_pVtx[4];           // 정점 데이터
...
```

```
m_pVtx[0] = VtxD(-0.9F, 0.9F, 0, D3DXCOLOR(1,0,0,1));
m_pVtx[1] = VtxD( 0.9F, 0.9F, 0, D3DXCOLOR(0,1,0,1));
m_pVtx[2] = VtxD( 0.9F, -0.9F, 0, D3DXCOLOR(0,0,1,1));
m_pVtx[3] = VtxD(-0.9F, -0.9F, 0, D3DXCOLOR(1,0,1,1));
```

고정 기능 파이프라인에서는 디바이스의 DrawPrimitive…() 함수를 호출하면 바로 출력되었습니다. 우리는 퍽셀 쉐이더를 이용해서 출력하는 것이 목표이기 때문에 다음과 같이 쉐이더 코드를 작성합니다.

```
ps_1_1          // 퍽셀 쉐이더 버전 선언
mov r0, v0      // 출력 레지스터에 복사
```

v0는 레스터 처리를 거친 Diffuse 값입니다. 퍽셀 쉐이더 버전 1.x에서는 임시 레지스터로 사용되는 r0가 출력 레지스터 이기도 합니다. 버전 1.1에서는 임시 레지스터를 r0와 r1 2개 정도만 사용할 수 있습니다.

이와 동등한 코드를 퍽셀 쉐이더 2.0로 작성하면 다음과 같습니다.

```
ps_2_0          // 퍽셀 쉐이더 버전 선언
dc1 v0          // Diffuse를 입력 레지스터 v0 선언
mov oC0, v0      // 출력 레지스터에 복사
```

dc1은 입력 레지스터를 선언할 때 사용되며 v#는 색상, 텍스처 좌표는 t#, 샘플러는 s#으로 지정합니다.

이 퍽셀 쉐이더 명령어를 컴파일하고 퍽셀 쉐이더 객체를 생성하는 코드는 다음과 같습니다.

```
DWORD dwFlags = 0;
#if defined( _DEBUG ) || defined( DEBUG )
    dwFlags |= D3DXSHADER_DEBUG;
#endif

LPD3DXBUFFER pShd = NULL;           LPD3DXBUFFER pErr = NULL;
hr = D3DXAssembleShaderFromFile( "data/Shader.psh"
    , NULL, NULL, dwFlags, &pShd, &pErr);

if( FAILED(hr) )
{
```

```

if(pErr)
{
    MessageBox( hWnd, (char*)pErr->GetBufferPointer(), "Err", MB_ICONWARNING);
    pErr->Release();
}
else
{
    MessageBox( hWnd, "File is Not exist", "Err", MB_ICONWARNING);
}
return -1;
}

hr = m_pDev->CreatePixelShader( (DWORD*)pShd->GetBufferPointer() , &m_pPs);
pShd->Release();

if ( FAILED(hr) )
    return -1;

```

정점 쉐이더와 마찬가지로 쉐이더 컴파일은 문자열은 D3DXAssembleShader()함수를 사용하고 파일은 D3DXAssembleShaderFromFile() 함수를 사용합니다. 퍽셀 쉐이더 객체는 디바이스의 CreatePixelShader() 함수를 이용해서 퍽셀 쉐이더 객체를 생성합니다.

이렇게 만든 퍽셀 쉐이더 객체를 렌더링에서 사용하기 위해서는 정점 쉐이더와 유사하게 퍽셀 처리를 프로그램 가능한 파이프라인 사용을 디바이스에 알려야 합니다.

```
m_pDev->SetPixelShader(m_pPs);
```

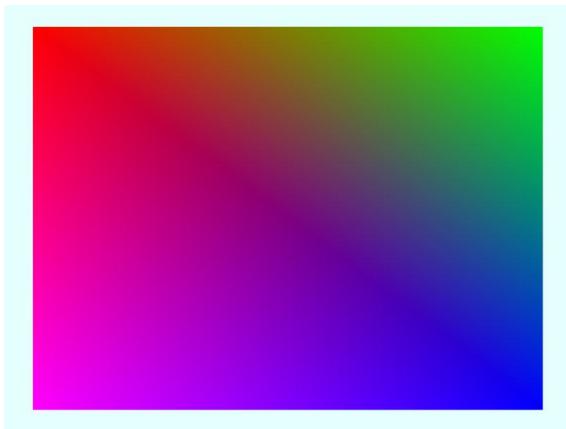
이후 과정은 렌더링은 고정 기능 파이프라인과 같습니다.

```
m_pDev->SetFVF(...);
m_pDev->DrawPrimitive(...);
```

프로그램 가능한 파이프라인 사용이 끝나면 이 또한 디바이스에게 알립니다.

```
m_pDev->SetPixelShader( NULL);
```

이와 관련한 전체 코드는 [s1p\\_01\\_basic1.zip](#)의 ShaderEx.cpp와 "data/shader.psh" 파일을 참고 하기 바랍니다.



<정점 색상 출력 퍽셀 쉐이더. [s1p\\_01\\_basic1.zip](#)>

### 3.1.2 상수 레지스터 설정

퍽셀 쉐이더에서 처리되는 데이터 또한 정점 쉐이더와 마찬가지로 float4형이고 색상에 대해서 r, g, b, a 또는 x, y, z, w로 접근할 수 있으며 swizzling 또한 가능합니다.

상수 설정도 float4형으로 전달해야 하며 SetPixelShaderConstantF() 함수를 사용합니다. 이 함수는 SetVertexConstantF()와 비슷하게 상수 레지스터의 값을 설정하는 함수로 첫 번째 인수는 상수 레지스터 번호를 지정하고, 두 번째 인수는 float형 데이터의 배열을 설정합니다. 마지막 세 번째 인수는 float4형의 개수를 정합니다.

상수 레지스터 설정의 우선 순위는 쉐이더 내부에서 정한 코드입니다. 정점 쉐이더와 마찬 가지로 외부에서 SetPixelShaderConstantF() 함수로 상수 레지스터의 값을 설정해도 쉐이더 내부에서 정의되어 있으면 이 정의된 값으로 상수 레지스터가 정해집니다.

만약 Diffuse와 두 개의 색상을 곱해서 최종 색상 = Diffuse \* 색상 0 \* 색상 1으로 정하는 것을 퍽셀 쉐이더로 작성하면 다음과 같습니다.

```
ps_1_1
def c0, 2, 2, 2, 1      // 상수 레지스터 c0.rgb = (2,2,2,1)
mul r0, c0, v0          // 상수 값과 색상 혼합
mul r0, r0, c1          // 외부에서 정한 상수 값과 색상 혼합
```

또는

```
ps_2_0
def c0, 2, 2, 2, 1      // 상수 레지스터 c0.rgb = (2,2,2,1)
```

```

dcl v0           // Diffuse를 입력 레지스터 v0 선언
mul r0, c0, v0   // 상수 값과 Diffuse 합
mul r0, r0, c1   // 외부에서 정한 상수 값과 색상 혼합

mov oC0, r0       // 출력 레지스터에 복사

```

쉐이더 내부에서 상수 레지스터 c0가 설정되었습니다. 상수 레지스터를 외부에서 설정하려면 SetPixelShaderConstantF()함수로 다음과 같이 작성합니다.

```
D3DXCOLOR color0(0, 0, 0, 1);
```

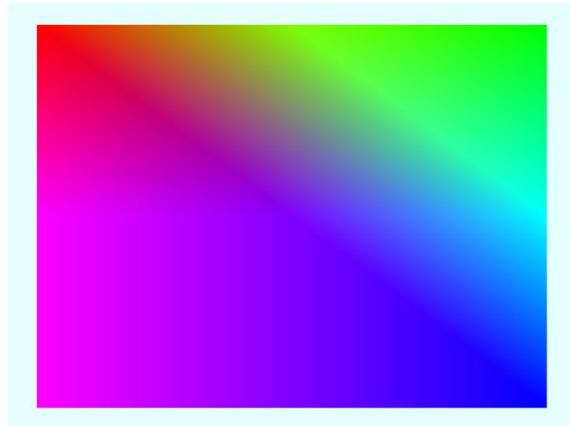
```
m_pDev->SetPixelShaderConstantF(0, (float*)&color0, 1); // 상수 레지스터 값 설정
```

```
D3DXVECTOR4 color1(0.5f, 1, 1, 1);
```

```
m_pDev->SetPixelShaderConstantF(1, (float*)&color1, 1); // 상수 레지스터 값 설정
```

색상에 대해서 D3DXVECTOR4 구조체를 사용할 수 있지만 D3DXCOLOR 구조체를 사용하는 것이 보기 좋습니다.

외부에서 c0를 설정하고 있지만 c0는 쉐이더 내부에서 "def"로 이미 설정되어 있어 이 값이 렌더링에 적용됩니다.



<픽셀 쉐이더 상수 레지스터 설정. [slp\\_01\\_basic2\\_const.zip](#)>

### 3.1.3 색상 반전(Invert)

색상 밝기의 반전(Invert)은 다음과 같이 계산 합니다.

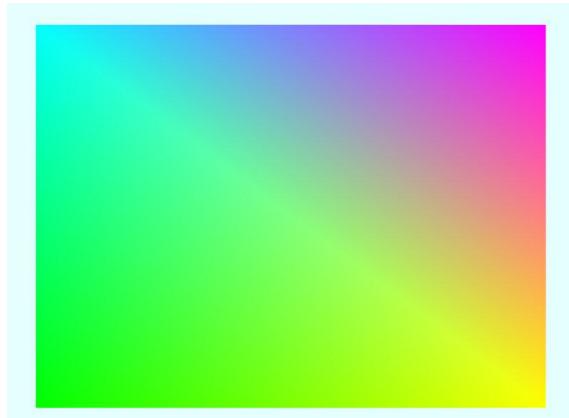
반전 색상 =  $1.0 - \text{색상}$

고정 기능 파이프라인에서 색상을 반전하려면 전체 픽셀을 복사해서 이 공식을 적용해야 하는데 픽셀 쉐이더를 사용하면 다음과 같이 간단한 코드로 사용하면 입력된 Diffuse에 대해서 색상을 반전 시킬 수 있습니다.

```
ps_1_1
def c0, 1, 1, 1, 0
sub r0, c0, v0
```

또는

```
ps_2_0
def c0, 1, 1, 1, 0
dc1 v0
sub r0, c0, v0
mov oC0, r0
```



<색상 반전(Invert): [s1p\\_01\\_basic3\\_invert.zip](#)>

### 3.1.4 색상 단색(Monotone)

R, G, B 색상을 하나의 색상으로 만드는 단색화(Monotone)도 픽셀 쉐이더를 사용하면 편리합니다.

$$\text{단색화 색상} = (R + G + B)/3$$

으로 정의할 수 있지만 사람의 눈은 Green과 Red에 더 민감하다고 합니다. 경험적인 데이터에 의해 단색화 색상은 다음과 같이 정의 합니다.

단색화 색상 = R \* 0.299 + G \* 0.587 + B \* 0.114

RGB 색상을 3차원 벡터로 생각하면 단색화 색상 공식은 내적을 이용한 것과 동일합니다.

단색화 색상 = `dot( (R, G, B), (0.299, 0.587, 0.114) )`

쉐이더 코드를 구성할 때 같은 기능이라면 쉐이더에서 제공하는 함수를 사용하고, 저 수준의 경우 한 줄이라도 덜 작성하는 것이 효율이 높다고 합니다. 퍽셀 쉐이더에서도 내적에 대한 `dp3`, `dp4` 연산자가 있어서 단색화 색상을 다음과 같이 작성할 수 있습니다.

```
ps_1_1
def c0, 0.299, 0.587, 0.114, 0
dp3 r0, c0, v0
```

또는

```
ps_2_0
def c0, 0.299, 0.587, 0.114, 0
dcl v0
dp3 r0, c0, v0
mov oC0, r0
```



<단색화(Monotone): [s1p\\_01\\_basic4\\_mono.zip](#)>

단색화는 게임에서 자주 사용되는 기술 이므로 꼭 기억하기 바랍니다.

## 3.2 Texturing & Multi-Texturing

픽셀 쉐이더에서 텍스처 색상을 처리하는 방법은 다른 픽셀 처리(Pixel Processing)과 동일하며 텍스처에서 색상을 가져오는 샘플링(Sampling)만 추가될 뿐입니다. 그런데 픽셀 쉐이더 1.x 은 텍스처 색상을 가져오는 명령어들이 버전마다 차이가 있고 제조사마다 지원이 되지 않는 버전도 존재했었는데 이것은 픽셀 처리량이 정점 쉐이더 보다 많아 GPU를 만드는 가격과 기술의 문제가 있었기 때문이며 현재 대부분의 그래픽 카드는 픽셀 쉐이더 2.0 이상 지원 되고 있어서 이 버전부터 사용 방법과 명령어 구성의 일관성이 유지되고 있어서 이 버전 이상에서 작업하는 것이 좋습니다.

픽셀 쉐이더 1.1에서 텍스처 색상을 가져올 때는 tex 명령어와 t# 레지스터를 사용해서 "tex t#" 으로 작성합니다.

### ps\_1\_1

```
tex t0      // stage 0의 텍스처 좌표에서 샘플링
tex t1      // stage 1의 텍스처 좌표에서 샘플링
```

tex는 텍스처의 샘플링 명령어로 "tex t0"는 다중 텍스처 처리(Multi-Texturing)의 0 번째 Stage 텍스처 좌표에 해당하는 픽셀 R, G, B, A를 가져와 t0에 저장합니다. "tex t1"은 1 번째 Stage 텍스처 좌표의 픽셀을 t1에 저장합니다.

ATI 계열만 지원되는 픽셀 쉐이더 버전 1.4의 경우 texId를 사용합니다. texId는 두 개의 Operand 가 필요합니다.

### ps\_1\_4

```
texId r0, t0  // stage 0의 텍스처 좌표 t0에서 샘플링, r0에 저장
texId r1, t1  // stage 1의 텍스처 좌표 t1에서 샘플링, r1에 저장
```

texId의 첫 번째 Operand 임시 레지스터 r#은 픽셀의 저장 장소이고, 두 번째 Operand t#은 텍스처 좌표에 해당합니다.

앞의 "texId r0, t0"는 t0(0 번째 Stage 텍스처 좌표)에 해당되는 픽셀을 r0에 저장하고, "texId r1, t1"은 1 번째 텍스처 좌표(t1)의 픽셀을 r1에 저장하는 것입니다.

픽셀 쉐이더 2.0부터 텍스처 색상을 추출하는 샘플러 객체 "s#"와 임시 레지스터 r0 대신 출력 레지스터 "oC#"이 추가되었습니다. 그리고 문법 체계도 바뀌어서 정점 쉐이더 문법과 비슷하게 입력 레지스터를 모두 선언해야 합니다.

다음은 샘플러와 입력 레지스터를 선언하고 텍스처에서 색상을 추출하는 예입니다.

## ps\_2\_0

```

dcl_2d s0           // 2차원 텍스처에 대한 샘플러 선언
dcl    t0.xy         // 0번 Stage에 대한 2차원 xy 텍스처 좌표 선언
texld  r0, t0, s0    // 샘플러를 사용한 텍스처 색상 추출
mov    oC0, r0        // 출력 레지스터에 복사

```

샘플러 선언은 1차원 텍스처는 "dcl\_1d s#", 2차원 텍스처는 "dcl\_2d s#", 입방체(cube) 텍스처는 "dcl\_cube s#", 볼륨 텍스처는 "dcl\_volume s#"을 선언할 때 사용 합니다. 텍스처 좌표 지정은 "dcl t#"로 합니다. 1차원 텍스처 좌표는 "dcl t#.x", 2차원 텍스처 좌표는 "dcl t#.xy" 등 t# 레지스터 다음에 각 차원에 해당하는 좌표계만큼 x, y, z, w 순서대로 적습니다.

샘플러 레지스터의 번호는 고정 기능 파이프라인에서 디바이스의 SetTexture() 함수 첫 번째 인수의 Stage에 해당합니다.

텍스처에서 색상을 가져오는 샘플링 명령은 texld를 사용합니다. 2.0에서 texld는 3개의 Operand를 사용합니다. 첫 번째 Operand는 추출한 색상을 저장 장소이고, 두 번째 Operand는 텍스처 좌표 레지스터 t#입니다. 세 번째 Operand는 샘플러 레지스터 s#입니다.

이렇게 샘플러, 텍스처 좌표 등이 분리되어 있어서 한 개의 텍스처 좌표가 입력되더라도 이것을 변화시켜가면서 샘플링 할 수 있어서 다중 텍스처 처리(Multi-Texturing), Post Effect 등에서 굀셀 쉐이더 2.0 이상을 사용하는 것이 작업하기에 편리합니다.

D3D 굀셀 쉐이더 2.0은 굀셀의 결과에 대해서 Multi-element에 대한 출력이 가능합니다. 출력 레지스터 oC#는 oC0 ~ 0C3까지 있으며 최소한 oC0 레지스터 출력이 설정되어야 쉐이더 컴파일이 됩니다.

지금까지 쉐이더 버전마다 텍스처 샘플링에 대해서 살펴보았습니다. 쉐이더를 사용한 다중 텍스처 처리는 색상의 연산이므로 샘플링에 대한 방법만 알면 간단하게 만들 수 있습니다.

다음은 텍스처의 색상을 그대로 화면에 출력하는 쉐이더 코드입니다.

## ps\_1\_1

```

tex t0           // 텍스처 좌표 t0에 대한 샘플링
mov r0, t0        // 출력 레지스터에 복사

```

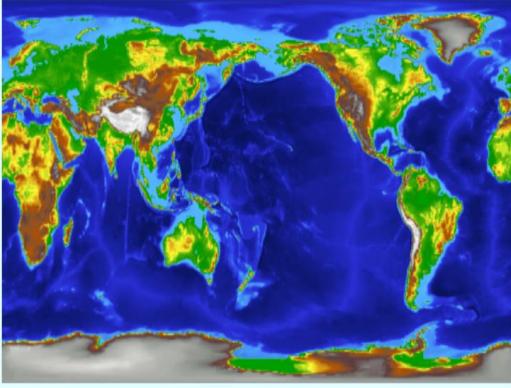
또는

## ps\_2\_0

```

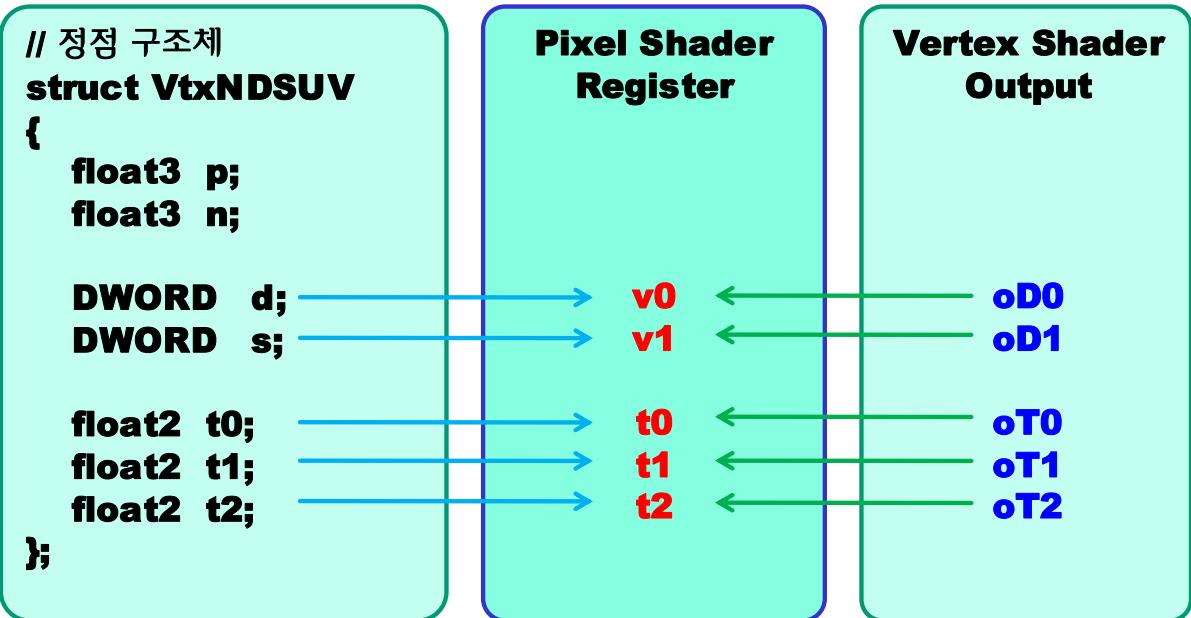
dcl_2d s0           // 2D 샘플러 선언
dcl    t0.xy         // 2차원 텍스처 좌표 선언
texld  r0, t0, s0    // 샘플링
mov    oC0, r0        // 출력 레지스터에 복사

```



<단순 텍스처 색상 출력: [s1p\\_02\\_1tex1.zip](#)>

대부분 정점 처리의 Diffuse 또는 Specular 값과 혼합해서 출력합니다. 퍽셀 쉐이더에서 정점 처리의 Diffuse와 Specular는 입력 레지스터 v0, v1으로 선언하고 텍스처 좌표는 t#으로 선언해서 사용합니다. 정점 구조체와 정점 쉐이더 출력 레지스터, 퍽셀 쉐이더 입력 레지스터의 관계를 비고 하면 다음 그림과 같습니다.



<정점 구조체, 정점 쉐이더 출력 레지스터, 퍽셀 쉐이더 입력 레지스터와의 관계>

간단하게 다중 텍스처 처리의 색상 연산 MODULATE4X 를 쉐이더로 구현해봅시다. MODULATE4X 는 ARG1과 ARG2를 곱하고 다시 4배를 합니다.

MODULATE4X 최종 색상 = 정점 처리 Diffuse \* 텍스처 색상 \* 4

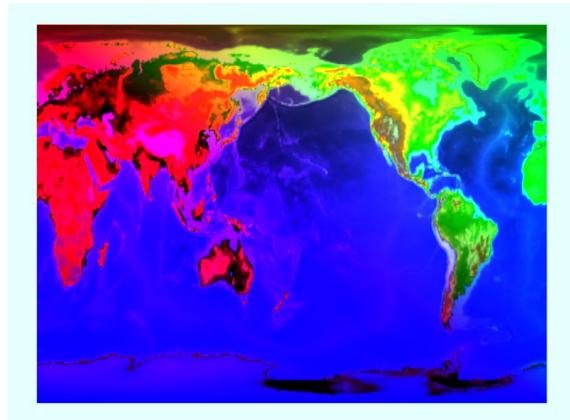
```
ps_1_1
tex t0           // 0번 째 Stage의 텍스처를 t0에 샘플링
mul_x4 r0, t0, v0 // 출력 색상 = 텍스처 * Diffuse * 4
```

또는

```
ps_2_0
def    c31, 4, 0, 0, 0
dcl    v0          // Diffuse 선언
dcl_2d s0          // 2D 샘플러
dcl    t0.xy        // 0번 Stage의 텍스처 좌표

texld r0, t0, s0    // 샘플링
mul r0, r0, v0      // 색상 = 텍스처 * Diffuse
mul r0, r0, c31.x   // 색상 *= 4
mov oC0, r0          // 출력
```

1.X 버전에서는 수정자(modifier) modifier가 있어서 명령문을 줄일 수 있습니다. 앞의 mul\_x4는 mul 연산과 이 연산의 결과에 4배를 지시하는 것입니다. 나누기는 \_d2, \_d4, \_d8 등이 있지만 사용을 잘 안 합니다. 또한 결과의 범위를 [0, 1]로 한정하는 \_sat(Saturation)도 있습니다. 그런데 아쉽게도 2.0부터 이를 수정자는 사용할 수 없고 상수를 설정해서 직접 계산해야 합니다.



<멀티 텍스처 MODULATE4X: [slp\\_02\\_1tex2.zip](#)>

고정 기능 파이프라인은 색상 처리 방식이 몇 가지로 한정되어 있지만 퍽셀 쉐이더를 사용하면 처리 방식이 자유롭고 쉽게 코드로 만들 수 있습니다. 예를 들어 다음과 같이 최종 색상을 만든다고 합시다.

최종 색상 = 외부 색상 \* Diffuse \* 텍스처 색상 + 베이스 색상

이것을 고정 기능 파이프라인에서 구현하려면 3개 이상의 다중 처리(Multi-Texturing)을 거쳐야 합니다. 하지만 이것을 쉐이더 코드로 작성하면 다음과 같이 간단하게 만들 수 있습니다.

ps\_1\_1

```
def c0, 0.2, 0.2, 0.2, 0           // 베이스 색상

tex t0                           // 0번 Stage의 텍스처를 t0에 샘플링

mul r0, c1, v0                   // 외부 상수 값과 Diffuse 곱셈
mul r1, r0, t0                  // r1 = Diffuse * c1 * texture 색상
add r0, r1, c0                  // 출력= r1 + c0
```

또는

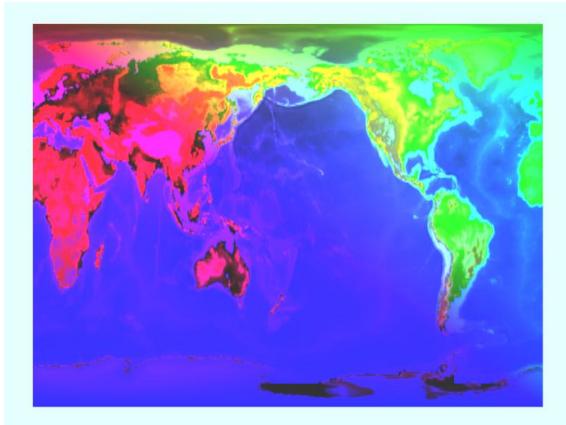
ps\_2\_0

```
def c0, 0.2, 0.2, 0.2, 0

dc1      v0
dc1_2d  s0                      // 2D 샘플러
dc1      t0.xy                   // 0번 Stage의 텍스처 좌표

tex1d r0, t0, s0                // 샘플링

mul r1, c1, v0                  // r1 = 외부 상수 값* Diffuse
mul r1, r1, r0                  // r1 = Diffuse * c1 * texture 색상
add r0, r1, c0                  // 색상 = r1 + c0
mov oC0, r0                      // 출력
```



<다중 텍스처 처리(Multi-Texturing): [slp\\_02\\_1tex3.zip](#)>

두 개의 텍스처의 혼합도 쉐이더를 사용하면 무척 편리합니다. 입력된 두 텍스처의 색상을 더하고 Diffuse와 곱해서 최종 색상을 출력하도록 쉐이더를 작성하면 다음과 같이 1.1에서 총 5줄이면 충분합니다.

```
ps_1_1
tex t0           // 0번 째 Stage 텍스처 샘플링
tex t1           // 1번 째 Stage 텍스처 샘플링
add r0, t0, t1   // 두 텍스처 색상을 더함
mul r0, r0, v0   // 더한 색상에 Diffuse를 곱하고 출력 레지스터 r0에 복사
```

만약 1번 Stage에 대한 텍스처 좌표를 0번 Stage의 좌표를 사용하려면 다중 처리 상태를 다음과 같이 설정합니다.

```
m_pDev->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0);
```

1.4 버전은 다음과 같이 texId를 사용하고 텍스처 좌표를 지정할 수 있어서 같은 텍스처 좌표를 사용할 때 1.1 버전처럼 디바이스의 다중 텍스처 처리 상태를 설정할 필요가 없습니다.

```
ps_1_4
texId r0, t0      // 0 번째 텍스처 좌표에 해당하는 0번 째 텍스처 샘플링
texId r1, t0      // 0 번째 텍스처 좌표에 해당하는 1번 째 텍스처 샘플링
add r2, r0, r1    // 두 텍스처 색상을 더함
mul r0, r2, v0    // 더한 색상에 Diffuse를 곱하고 출력 레지스터 r0에 복사
```

2.0은 Diffuse, 샘플러, 텍스처 좌표를 전부 선언해야만 사용할 수 있기 때문에 이 부분에 대해서만 코드가 길어질 뿐입니다.

```

ps_2_0
dcl    v0
dcl_2d s0           // 2D 셈플러 0
dcl_2d s1           // 2D 셈플러 1
dcl    t0.xy         // Texture Coordinate at stage 0
tex1d r0, t0, s0    // 0번 텍스처 좌표에 해당하는 0번 텍스처 셈플링
tex1d r1, t0, s1    // 0번 텍스처 좌표에 해당하는 1번 텍스처 셈플링
add   r2, r0, r1    // 두 텍스처 색상을 더함
mul   r0, r2, v0    // 더한 색상에 Diffuse를 곱함
mov   oC0, r0        // 출력

```



<Multi-Texturing: [slp\\_02\\_2tex\\_multi.zip](#)>

### 3.3 단색 화면(Monotone Effect)

만약 쉐이더를 사용하지 않고 출력 색상을 단색으로 만들고자 한다면 여러분은 디바이스의 후면 버퍼를 구성하는 색상버퍼에서 픽셀을 가져와 단색으로 만들어야 합니다.

```

IDirect3DSurface9*      pSrc;
hr = m_pd3DDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &pSrc);
...
hr = pSrc->LockRect(&rc, NULL, 0);
DWORD* pColor = (DWORD*)rc.pBits;

for(int i=0; i<dsc.Width* dsc.Height; ++i)
{

```

```

D3DXCOLOR      color = pColor[i];
FLOAT         d = color.r * 0.299f + color.g * 0.587f + color.b * 0.114f;
pColor[i] = D3DXCOLOR(d,d,d,1);
}

```



<화면 전체에 대한 단색화: [s1p\\_03\\_mono0\\_fixed.zip](#)>

전체 화면을 단색으로 몇지게 처리했지만 가장 큰 문제는 프레임 속도입니다. 이 방식의 문제는 어떤 하드웨어 가속도 지원이 되지 않는 방식이라서 렌더링 속도가 빨라야 ~10 FPS 정도여서 게임에서 사용하기는 부적합니다.

쉐이더를 사용하면 렌더링 속도의 저하 없이 화면 전체를 단색으로 만들 수 있습니다. 방식은 다음 그림처럼 3D 장면을 텍스처에 저장하고 이 텍스처를 화면 영역과 동일한 4개의 정점에 매핑 한 다음 다시 디바이스에 출력하는 것입니다.



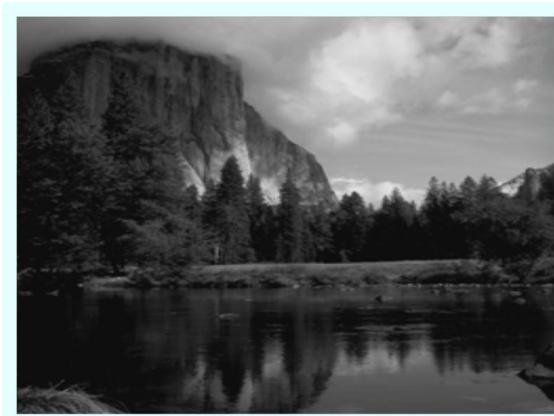
<화면 단색화 방법>

픽셀을 하나의 색상으로 만드는 단색화 방법은 이전에 살펴 보았고 이것을 다음과 같은 공식으로 표현할 수 있음을 우리는 알고 있습니다.

```
단색화 색상 = dot( (R, G, B), (0.299, 0.587, 0.114) )
```

이것을 텍스처에 적용하면 다음과 같이 작성 할 수 있습니다.

```
ps_2_0
def c0, 0.299, 0.587, 0.114, 0
dcl    v0
dcl_2d s0           // 2D 샘플러 0
dcl    t0.xy        // 텍스처 좌표
tex1d r0, t0, s0   // 샘플링 0
dp3   r0, r0, c0   // 내적으로 단색 색상 생성
mov    oC0, r0      // 출력
```



<텍스처 단색화 출력: [slp\\_03\\_mono1.zip](#), [slp\\_03\\_mono2.zip](#)>

다음으로 화면 전체를 실시간으로 만든 텍스처에 렌더링 해야 하는데 이것은 서피스 효과 강좌에서 만들었던 IrenderTarget 객체를 이용하겠습니다. IrenderTarget 사용은 다음과 같습니다.

```
IrenderTarget* m_pTrnd;

CMain::Init()
...
// 렌더 타깃 용 텍스처 생성
if(FAILED(LcD3D_CreateRenderTarget(...)))
    return -1;
```

```
CMain::FrameMove()
```

```

...
// 렌더 타깃에 장면 그리기
m_pTrnd->BeginScene();
    this->RenderScene();
m_pTrnd->EndScene();

CMain::Render()
...
// 쉐이더 실행, 전체 화면에 다시 그리기
LPDIRECT3DTEXTURE9      pTx = (LPDIRECT3DTEXTURE9)m_pTrnd->GetTexture();
m_pShader->SetSceneTexture(pTx);
SAFE_RENDER(   m_pShader   );

```

색상을 곱하면 어두워지므로 곱셈의 단색화 색상에 대한 색상 비중 값을 좀 더 밝게 올리고 외부에서 색상을 조정할 수 있도록 쉐이더를 작성하면 다음과 같습니다.

```

ps_1_1
def c0, 0.8, 0.9, 0.4, 0      // 색상 비중 값
tex t0                      // 0 번 스테지 텍스처 샘플링
dp3 r0, t0, c0                // 내적으로 간단히 단색 만들
mul r0, r0, c1                // 외부 상수와 곱해서 최종 색상 출력

```

또는

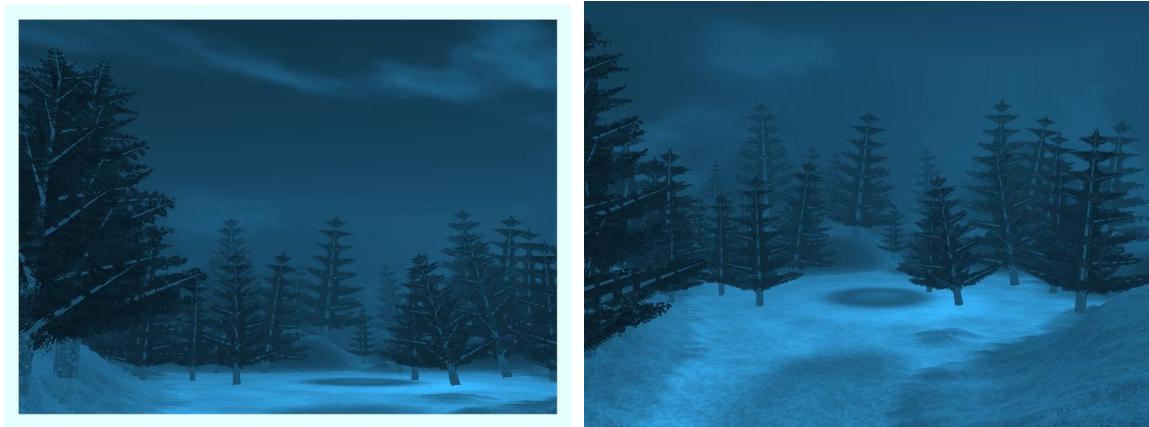
```

ps_2_0
def c0, 0.8, 0.9, 0.4, 0
dcl    v0
dcl_2d s0                      // 2D 샘플러 0
dcl    t0.xy                     // 텍스처 좌표 0

texld  r0, t0, s0              // 샘플링

dp3 r0, r0, c0                // 내적으로 단색 색상 생성
mul r0, r0, c1                // 외부 상수와 곱해서 최종 색상 출력
mov oC0, r0                    // 출력

```

<단색화 화면: [s1p\\_03\\_mono3\\_shader.zip](#)>

### 3.4 Blur 효과

흐림(Blur) 효과는 단색 효과와 마찬가지로 픽셀 쉐이더를 사용하는 대표적인 예입니다. 흐림 효과는 Gaussian blur를 많이 사용하지만 여기서는 단순하게 픽셀에 인접한 좌, 우, 상, 하 픽셀들과 자기 자신을 더한 값의 평균을 최종 색상으로 정하는 방식을 구현해 보겠습니다.

샘플링은 자신을 포함해서 좌, 우, 상, 하 총 5번 진행되기 때문에 좌표 또한 정점의 구조체는 총 5개의 텍스처 좌표를 가지고 있어야 합니다.

```
struct VtxDUV1
{
    VEC3      p;
    DWORD     d;
    FLOAT     u0,v0;
    FLOAT     u1,v1;
    FLOAT     u2,v2;
    FLOAT     u3,v3;
    FLOAT     u4,v4;
};
```

이 구조체에 각각의 uv 좌표를 설정한 다음과 같이 쉐이더 코드를 작성합니다.

```
ps_1_4
def c0, 0, 0, 0, 0.2
tex1d r0, t0
tex1d r1, t1
```

```

texId r2, t2
texId r3, t3
texId r4, t4
add r5, r0, r1
add r5, r5, r2
add r5, r5, r3
add r5, r5, r4
mul r1, r5, c0.www
mov r0, r1

```

렌더링에서 같은 텍스처를 여러 Stage에 연결합니다.

```

for(i=0; i<5; ++i)
    m_pDev->SetTexture(i, m_pTx);

```

이렇게 설정한 다음 렌더링 하면 흐림 효과를 만들어 낼 수 있으며 전체 코드는 [s1p\\_04\\_blur1.zip](#)를 참고 하기 바랍니다.

픽셀 쉐이더 2.0 이상을 사용하면 텍스처 좌표를 변화해 가며 샘플링 할 수 있습니다. 다음의 쉐이더 코드에서 상수 값은  $3.5f/800.f$ ,  $3.5f/256.f$  값으로 바로 인접한 픽셀이 아닌 3.5만큼 떨어져 있는 픽셀을 샘플링하기 위한 값입니다.

```

ps_2_0
def c0, 0.0,-0.004375f, 0.0, 0.2
def c1, 0.0, 0.004375f, 0.0, 0.0
def c2, -0.005833f, 0.0, 0.0, 0.0
def c3, -0.005833f, 0.0, 0.0, 0.0

dcl_2d s0           // 2D 샘플러 0
dcl     t0.xyzw      // 텍스처 좌표 0
add    r0, c0, t0      // 텍스처 좌표를 왼쪽(0, -3.5)으로 이동
texId r0, r0, s0      // Sampling texcoord (0, -3.5)
add    r1, c1, t0
texId r1, r1, s0      // Sampling texcoord (0, +3.5)
add    r2, c2, t0
texId r2, r2, s0      // Sampling texcoord (-3.5, 0)
add    r3, c3, t0

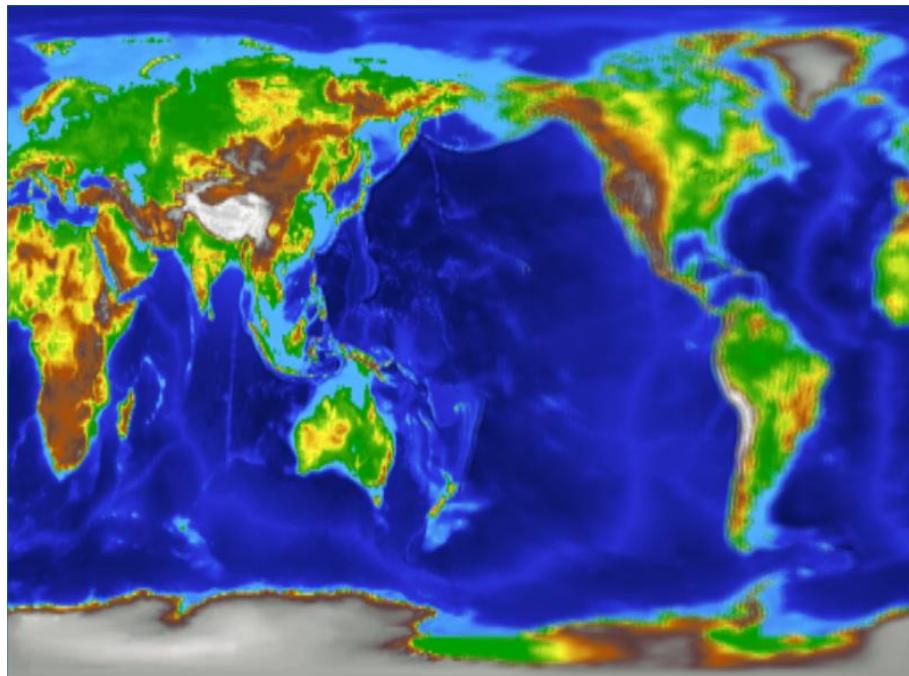
```

```

texld r3, r3, s0           // Sampling texcoord (+3.5, 0)
texld r4, t0, s0           // Sampling texcoord (0, 0)
add r0, r0, r1
add r0, r0, r2
add r0, r0, r3
add r0, r0, r4
mul r0, r0, c0.wwww
mov oC0, r0                // 출력

```

쉐이더 코드가 조금 길어졌는데 흐름 효과는 저 수준으로 작성하는 것보다 HLSL을 사용하는 것이 편리합니다. 저 수준은 "이렇게 하는 방법도 있구나" 하는 정도로만 기억하기 바랍니다.



<흐름 효과: [s1p\\_04\\_blur1.zip](#), [s1p\\_04\\_blur2.zip](#)>

### 3.5 Pixel Shader Effect

앞서 정점 쉐이더를 Wrapping 하기 위해서 ILcShader를 만들었습니다. 이 클래스를 픽셀 쉐이더에서도 사용할 수 있도록 수정해 봅시다. 클래스의 인터페이스 모양은 변하지 않고, 생성 함수에서 정점 쉐이더와 픽셀 쉐이더를 구분할 수 있도록 인수를 추가합니다.

```
// sCmd: Vertex Shader: "vs", Pixel Shader: "ps"
```

```
INT LcDev_CreateShaderFromFile(char* sCmd
    , ILcShader** pData, void* pDevice, char* sFile);
```

```
INT LcDev_CreateShaderFromString(char* sCmd
    , ILcShader** pData, void* pDevice, char* sString, INT iLen);
```

CreateShaderFromFile() 함수에서 정점 쉐이더 또는 퍽셀 쉐이더 타입을 결정하고 ILcShader 객체를 생성합니다.

```
INT LcDev_CreateShaderFromFile(char* sCmd, ILcShader** pData, void* pDevice, char* sFile)
...
if( 0 == _stricmp(sCmd, "vs"))
    p->SetShaderType(CLcShader::ELC_VS);
else if( 0 == _stricmp(sCmd, "ps"))
    p->SetShaderType(CLcShader::ELC_PS);
else{ delete p; return -1; }

if(FAILED(p->Create(pDevice, sFile)))
{
    delete p; return -1;
}

*pData = p;
return 0;
```

함수의 구현에서는 sCmd 값을 가지고 정점 쉐이더와 퍽셀 쉐이더를 구분해서 객체를 생성하는 CLsShader::Create() 함수를 정점 쉐이더 또는 퍽셀 쉐이더 객체를 생성할 수 있도록 다음과 같이 변경합니다.

```
INT CLcShader::Create(void* p1, void* p2, void* p3, void* p4)
...
// Vertex Shader
if(ELC_VS == m_nShader)
    hr = m_pDev->CreateVertexShader( (DWORD*)pShd->GetBufferPointer()
        , &m_pVs);
```

```
// Pixel Shader
else if(ELC_PS == m_nShader)
    hr = m_pDev->CreatePixelShader( (DWORD*)pShd->GetBufferPointer()
        , &m_pPs);
```

쉐이더 상수를 설정하는 SetMatrix, SetVector, SetColor, SetFloat 함수도 정점 쉐이더와 퍽셀 쉐이더 둘 다 지원할 수 있도록 수정합니다.

```
INT CLcShader::SetVector(INT uReg, const D3DXVECTOR4* v)
{
    if(ELC_VS == m_nShader)
        return m_pDev->SetVertexShaderConstantF( uReg , (FLOAT*)v , 1);
    return m_pDev->SetPixelShaderConstantF( uReg , (FLOAT*)v , 1);
}
```

```
INT CLcShader::SetColor(INT uReg, const D3DXCOLOR* v)
{
    if(ELC_VS == m_nShader)
        return m_pDev->SetVertexShaderConstantF( uReg , (FLOAT*)v , 1);
    return m_pDev->SetPixelShaderConstantF( uReg , (FLOAT*)v , 1);
}
```

```
INT CLcShader::SetFloat(INT uReg, const FLOAT* v)
{
    if(ELC_VS == m_nShader)
        return m_pDev->SetVertexShaderConstantF( uReg , (FLOAT*)v , 1);
    return m_pDev->SetPixelShaderConstantF( uReg , (FLOAT*)v , 1);
}
```

Begin()/End() 함수도 같은 방식으로 정점 쉐이더, 퍽셀 쉐이더 지원이 되도록 수정합니다. 전체 코드는 다음 예제를 참고하기 바랍니다.

[s1p\\_05\\_IShaderEffect.zip](#)

### 3.6 Post Effect Example

앞서 픽셀 쉐이더 2.0 이상은 샘플러, 텍스처 좌표 등이 분리되어 있어서 쉐이더 코드에서 같은 텍스처 좌표에 편차(Deviation)를 주어 흐림 효과 등을 만들 수 있다고 했습니다. 흐림 효과 등 인접 픽셀을 처리할 때 미리 계산된 값들을 주변 픽셀에 가중치(또는 비중: Weight)을 주어 이 가중치에 각 픽셀을 곱하고 곱해진 픽셀들을 다시 더해서 최종 색상을 결정합니다. 인접한 픽셀에 가중치를 설정하는 것을 픽셀 마스킹이라 합니다. 이 마스킹 값에 따라 흐림 효과 같은 적분 형태가 될 수 있고, 색의 변화 부분에서 날카롭게 만드는 미분 형태도 존재합니다. 특히 미분 형태 중에서 장면의 외곽선 추출은 게임 프로그램에서 자주 응용되는 기술입니다.

이전에 구현했던 흐림 효과에 대해서 마스킹 테이블을 구성하면 다음과 같습니다.

0.0, 0.2, 0.0
0.2, 0.2, 0.2
0.0, 0.2, 0.0

<흐림 효과 마스킹 테이블>

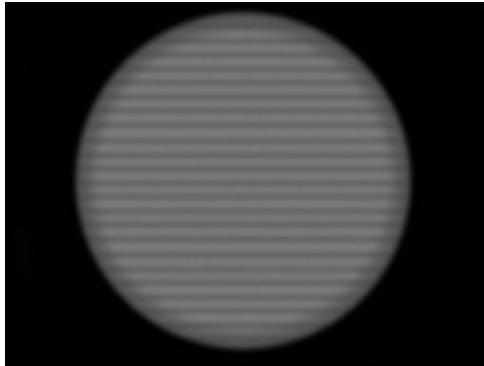
때로는 주변 픽셀의 가중 값을 음수(-) 값으로 설정 할 수 있습니다. 결과적으로 픽셀 사이의 빨름이 수행되는데 이러한 형태는 미분과 흡사하며 외곽선 등 픽셀의 변화가 큰 부분을 표현할 때 사용됩니다.

1, -1, 1
-1, 7, -1
1, -1, 1

<외곽선 마스킹 테이블>

이 외곽선 마스킹 테이블을 가지고 야간 투시경(Night Scope)을 구현해 봅시다. 가장 먼저 구현해야 할 것은 화면 전체 장면을 픽셀에 저장하는 것입니다. 이것은 이전의 단색화 과정에서 만들어 보았으므로 생략하겠습니다.

다음으로 다음과 같은 스코프 이미지가 필요합니다.



&lt;Scope 이미지&gt;

화면 전체를 저장한 픽셀은 마스킹 테이블의 값에 따라 총 9번 텍스처 좌표를 변화해 가면서 샘플링하고 가중치를 곱한 이 값을 더해서 색상을 만듭니다. 그리고 최종 출력 픽셀을 스코프 이미지와 곱셈으로 결정하면 야간 투시경이 만들어 집니다.

이를 쉐이더 코드로 작성하면 다음과 같습니다.

```

ps_2_0
def c10, 0.0, 1.0, 0.8, 0      // Min, Max, 전체 밝기
def c11, 0.7, 0.9, 0.3, 0      // 색상 비중 값

dcl    t0          // t0 텍스처 좌표 선언
dcl_2d s0          // 0-stage 샘플러 객체 선언
dcl_2d s1          // 1-stage 샘플러 객체 선언

// Circle Image
mov r0, t0          // 샘플링 1-stage with 0 stage texture coordinate
texld r0, r0, s1

// Render Target Image Sampling
mov r1, t0
add r1, r1, c0
texld r1, r1, s0

mov r2, t0
add r2, r2, c0
texld r2, r2, s0
...
// Multiple Masking Value

```

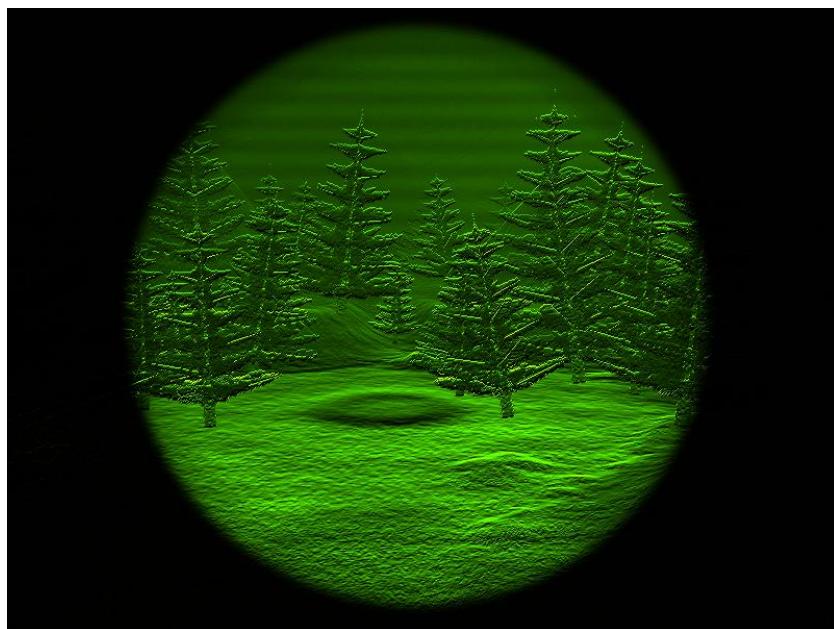
```

mul r1, r1, c20.x
mul r2, r2, c20.y
...
// Addl all Pixel
mov r10, r1
add r10, r10, r2
add r10, r10, r3
add r10, r10, r4
...
max r10, r10, c10.r      // (-)제거

dp3 r10, r10, c11      // 내적으로 간단히 단색 만들
mul r10, r10, c16      // 외부 상수와 곱해서 최종 색상 출력
mul r10, r0, r10        // Multiple r0, r10
mul r10, r10, c10.zzz  // 전체 밝기 조정
mov oC0, r10

```

이 쉐이더 코드는 [s1p\\_05\\_scope.zip](#)의 "data/shader.psh"에 구현되어 있고 실행하면 다음과 같은 화면을 볼 수 있습니다.



<저 수준 쉐이더 응용 - Night Scope: [s1p\\_05\\_scope.zip](#)>

저 수준 작성이 여러모로 힘이 많이 드니까 여러분은 HLSL을 이용하기 바랍니다.

## 4 High Level Shading Language 기초

HLSL(High Level Shading Language)은 단어 그대로 쉐이더에 대한 고 수준 언어입니다. 어셈블리 명령어 형식으로 작성한 정점 쉐이더와 텍셀 쉐이더 코드는 사용자에게 언어의 사용, 이해와 더불어 가독성과 융통성에 많은 부담을 줍니다. HLSL은 저 수준 언어인 쉐이더 명령어를 C나 파스칼과 같이 고 수준의 언어로 대신하게 되어 저 수준 언어의 어려운 부분들을 해소시켜줍니다.

고수준 언어로 작성된 쉐이더 프로그램은 컴파일 과정에서 필요한 명령어들을 컴파일러가 자동으로 추가합니다. 마이크로소프트에서 HLSL으로 작성한 고 수준 언어를 컴파일 해서 만든 명령어와 저 수준 쉐이더 명령어 만으로 작성한 명령어들을 비교한 자료를 발표한 적이 있는데 이 발표에 의하면 HLSL을 사용하더라도 둘의 차이가 크지 않음을 보인다고 합니다. 따라서 HLSL을 사용한 것과 직접 쉐이더 명령어를 사용한 결과는 차이가 없다라고 볼 수 있으므로 사용자는 HLSL을 이용하는 것이 가독성, 편리성 등 여러모로 유리한 점이 많이 있습니다.

<저 수준으로 작성한 정점 쉐이더>

```
vs_1_1
dcl_position    v0
dcl_color       v1
m4x4 oPos, v0, c0
mov  oD0, v1
```

<HLSL로 작성한 정점 쉐이더>

```
float         m_Bright;
float4x4      m_mtWVP;
struct SvsOut
{
    float4 Pos : POSITION;
    float4 Dif : COLOR0;
};

SvsOut VtxPrc(float3 Pos : POSITION, float4 Dif : COLOR0 )
{
    SvsOut Out = (SvsOut)0;
    Out.Pos = mul(float4(Pos, 1), m_mtWVP);
    Out.Dif = Dif * m_Bright;
    return Out;
}
```

두 코드를 비교해 보면 어셈블리 형식의 저 수준으로 작성한 쉐이더 코드가 훨씬 간결한 것을 알

수 있습니다. 하지만 HLSL로 작성한 것이 가독성이 빠르고 사용이 편리해 보입니다. 여러분이 저 수준 형식을 사용한다면 연산이 많아질수록 오히려 손이 더 가는 형태가 되어 불편합니다. 예를 들어 상수 설정이 많아지면 c0, c1 이러한 이름을 사용하는 것보다 "m\_mtWVP"와 같이 사용자가 직관적으로 정의한 변수를 사용하는 것이 좋고, 또한 함수도 만들어서 사용 할 수 있어서 저 수준 형식보다 HLSL이 장점이 많습니다. 그리고 이후에 알게 되겠지만 저 수준으로 작성하는 것이 꼭 코드의 길이가 짧아지는 것은 아니라는 것을 알게 될 것입니다.

여러분이 HLSL을 빨리 익히는 방법은 이전 강의에서 저 수준 언어로 작성된 코드를 HLSL로 바꾸어 보는 것이 가장 무난하지만 꼭 저 수준 언어를 알아야 HLSL을 제대로 이해하는 것은 아닙니다. 보통 HLSL은 100 Line 정도에서 작성되므로 만약 여러분이 C언어에 익숙하다면 HLSL에 대한 문법과 함수들을 연습하면 며칠 이내로 배울 수 있습니다. HLSL 함수 또한 걱정할 것 없는 것이 생각만큼 많은 숫자도 아니고 대부분 수학과 관련된 함수들이고 직관적으로 이해가 되기 때문에 HLSL로 장면에 대한 간단한 표현부터 연습하면 몇 일 이내로 쉽게 배울 수 있습니다.

## 4.1 Simple HLSL

HLSL은 저 수준 작성과 마찬가지로 문자열 또는 파일로 작성할 수 있습니다. 예를 들어 다음과 같이 정점의 위치를 출력하는 쉐이더 코드를 저 수준과 HLSL로 작성해 봅시다.

<저 수준>

```
"vs_1_1
dcl_position v0
mov oPos, v0
```

<HLSL>

```
float4 VtxPrc(float3 Pos : POSITION) : POSITION
{
    return float4(Pos, 1);
}
```

저 수준 언어를 기계어로 번역하는 것을 Assemble이라 하고 고 수준 언어는 Compile이라 합니다. 저 수준으로 작성한 쉐이더는 D3DXAssembleShader() 함수를 사용했습니다. 이와 대응되는 HLSL 함수는 D3DXCompileShader() 함수입니다. HLSL 문법으로 작성한 쉐이더 코드는 다음과 같이 컴파일 합니다.

```
LPD3DXBUFFER pShd = NULL;
```

```

LPD3DXBUFFER pErr = NULL;
hr = D3DXCompileShader(sHlsl, iLen
    , NULL, NULL
    , "VtxPrc" // 시작 함수
    , "vs_1_1" // 쉐이더 버전
    , dwFlags
    , &pShd // 컴파일 쉐이더
    , &pErr // Error
    , &m_pTbl // 상수 테이블
);

```

C언어로 작성한 응용프로그램은 int main() 함수가 필요합니다. HLSL은 사용자 함수를 지원하기 때문에 D3DXCompileShader() 함수의 4 번째 문자열 인수에 쉐이더 실행의 시작 함수를 지정해야 합니다. 또한 정점 쉐이더 또는 퍽셀 쉐이더 버전을 설정해야 하며, 상수 레지스터에 데이터 연결을 담당하는 상수 테이블을 생성할 수 있도록 상수 테이블 주소를 지정합니다. 파일에서 작성한 HLSL은 D3DXCompileShaderFromFile() 함수를 사용 합니다.

D3DXCompileShader() 함수는 단지 HLSL로 작성한 쉐이더 코드를 컴파일 하거나 실패에 대한 문법 에러를 검사하는 기능만 수행하기 때문에 컴파일 한 결과를 메모리에 적재하기 위해서 정점 쉐이더 또는 퍽셀 쉐이더 객체를 생성합니다. 이 부분은 저 수준과 동일합니다.

```

m_pDev->CreateVertexShader( (DWORD*)pShd->GetBufferPointer() , &m_pVs);
m_pDev->CreatePixelShader( (DWORD*)pShd->GetBufferPointer() , &m_pPs);

```

HLSL은 쉐이더 작성을 고급 언어로만 하겠다는 뜻이므로 저 수준과 마찬가지로 렌더링 파이프라인에 적용되는 정점 스트림에 대한 정점 선언자(Vertex Declarator)를 만들어야 합니다.

```

D3DVERTEXELEMENT9 vertex_decl[MAX_FVF_DECL_SIZE]={0};
D3DXDeclaratorFromFVF(Vtx::FVF, vertex_decl);
m_pDev->CreateVertexDeclaration( vertex_decl, &m_pFVF );

```

렌더링에서 프로그램 가능한 파이프라인을 사용하는 방법은 저 수준과 거의 동일하며 쉐이더의 전역 변수 설정은 컴파일 할 때 생성한 상수 테이블 객체를 이용합니다.

1. 정점 쉐이더 또는 퍽셀 쉐이더 사용을 명시함으로써 프로그램 가능한 파이프라인 사용을 알린다.
2. 정점 선언자를 디바이스에 연결한다.
3. 정점 선언 객체로 정점 데이터의 형식을 파이프라인에 알린다.

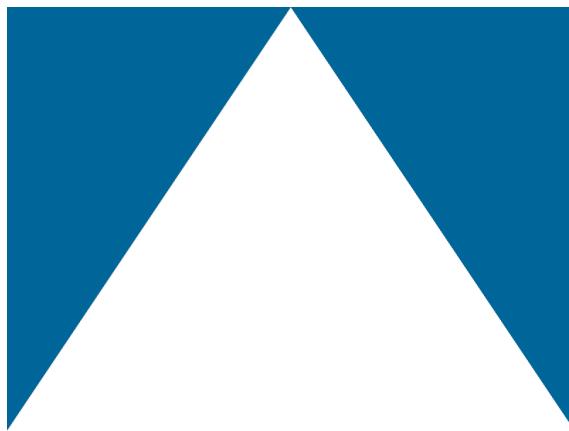
4. 상수 테이블을 사용해서 쉐이더의 전역 변수 값을 설정한다.
5. 정점을 그린다.
6. 정점 쉐이더 또는 픽셀 쉐이더 객체 사용을 해제해서 고정 기능 파이프라인 사용으로 돌아온다.

```
m_pDev->SetVertexShader(...);
m_pDev->SetPixelShader(...);
m_pDev->SetVertexDeclaration(...);
```

```
m_pDev->DrawPrimitiveUP(...);
```

```
m_pDev->SetVertexDeclaration(NULL);
```

```
m_pDev->SetVertexShader(NULL);
m_pDev->SetPixelShader(NULL);
```



<간단한 HLSL: [hs1\\_basic1.zip](#)>

## 4.2 HLSL 자료 형

Assembly 형식의 저 수준 쉐이더 언어는 사용법은 약간 까다롭지만 문법이 간단해서 프로그래머가 기억해야 할 내용이 많지 않았습니다. 하지만 HLSL은 C언어와 같은 고급 언어로 구성되어 있어서 문법에 대한 지식이 필요합니다.

저 수준 쉐이더에서 모든 데이터는 저장 장소 레지스터 이름을 그대로 사용해서 자료 형이라는 것이 필요하지 않았지만 HLSL은 프로그래머가 사용하는 변수 또는 상수를 컴파일러가 적절히 변역할 수 있도록 자료 형을 사용 하며 HLSL의 기본 자료 형(Data Type)과 복합 형(Complex

Type)으로 구분합니다.

기본 자료 형은 크게 Scalar Type(형), Vector 형, Matrix 형으로 나눕니다.

Scalar Type은 단일 자료로 형태로 true/false만 가지는 bool 형, 32bit인 int 형, 16비트 부동 소수점 half형, 32비트 float 형, 64비트 부동소수점 double 형이 있습니다. 이중에서 가장 많이 사용되는 타입은 float 형과 int 형입니다. 때로는 GPU의 특성에 따라 half, double은 지원이 안될 수 있으므로 타입 결정이 어려우면 float 형을 사용하는 것이 가장 무난합니다.

Scalar 형을 사용하는 방법은 다음과 같습니다.

선언: `float f;`

선언과 동시에 초기화: `float f = 3.1f;`

문법이 C언어와 같습니다. Scalar형은 배열을 만들어 사용할 수 있습니다. 다음은 float형 배열을 만들고 초기화 방법과 데이터 접근에 대한 예입니다.

배열: `float f[3];`

배열 선언과 동시에 초기화: `float f[3] = {1.f, 2.f, 3.f};`

배열 접근: `f[2] = 10.f;`

Scalar Type의 연산은 C언어의 산술 연산 +, -, \*, /,  $\{+|-|*|/\}$ = 등이 가능하고 정수 형의 경우 단항 연산 ++, --을 사용할 수 있습니다.

Vector Type은 자료를 표현하기 위해서 2개 이상의 자료가 결합된 형태로 가장 기본적인 형태는 vector이며 vector는 float형 4개가 하나의 자료인 float4 형입니다. vector 형은 자료를 접근하는 방법이 x, y, z, w 또는 r, g, b, a를 이용하거나 임의 접근 연산자 []를 사용합니다. 만약 vector의 전체 성분을 교체할 때는 vector 생성자를 사용합니다.

vector 선언: `vector v;`

vector 선언과 동시에 초기화: `vector v = {1,1,0,1};`

vector 접근1: `v.y = 3.0f;` or `v.g = 5.0f;`

vector 접근2: `v[2]=1.0f;`

vector 생성자: `v= vector (1, 0, 1, 1);`

Vector Type은 Scalar 형에 숫자를 붙여 차원을 붙여서 사용하기도 합니다. 예를 들어 float 형에 대해서 float2, float3, float4 형이 있고, int 형에 대해서 int2, int3, int4 형이 있습니다. 이들은 `vector<type, dimension>` 형태와 동등합니다.

```
float2 ≡ vector<float, 2>
float3 ≡ vector<float, 3>
float4 ≡ vector<float, 4>
int3 ≡ vector<int, 3>
int4 ≡ vector<int, 4>
```

선언과 동시에 초기화: `vector<double, 4> v = {1., 2., 3., 4.};`  
 성분 접근1: `v[2] = 3.4;`

이들 숫자가 붙은 자료 형도 Vector Type 이므로 vector와 마찬가지로 vector 생성자를 이용하거나 자신의 Type에 대한 생성자를 사용합니다.

```
v = vector<float, 4>(1, 1, 0, 1);
v = float4(1, 1, 0, 1);
```

주의할 것은 Vector Type을 Scalar 변수 또는 상수로 설정하면 변수의 모든 값은 Scalar 값으로 설정이 됩니다.

```
float k = 0.3f;
float4 v;
v = k;           // v.x = v.y = v.z = v.w = k
v = 1.0f;        // v.x = v.y = v.z = v.w = 1.0f
```

Vector Type은 swizzling 가능합니다. 단, xyzw와 rgba의 혼용은 허용이 안됩니다. 예를 들어 "v.xg", "v.yr" 등은 swizzling이 안됩니다.

기본 자료 형인 Matrix Type은 행렬을 표현한 자료 형입니다. 행렬 자료 형은 matrix 키워드를 사용해서 행과 열의 수로 표현합니다.

```
matrix <type, row, column>; type: int, float
3x3 float 형 행렬: matrix<float, 3, 3>
```

vector와 마찬가지로 숫자로 행과 열을 표현하기도 합니다.

```
int{row}x{column}: int2x2, int3x3, int4x4, int4x3
float{row}x{column}: float2x2, float3x3, float4x4, float4x3
```

행렬의 성분 접근에 대해서 행 우선 지정은 "row\_major" 키워드를 사용하고 열 우선 지정의 경우는 "col\_major" 키워드를 사용합니다. 이들은 GPU 내부에서 처리되는 방식을 결정하는 것이기 때문에 연산의 결과에는 영향을 주지 않습니다. 만약 이를 지정이 없으면 default로 행 우선 순위 방식으로 행렬에 대해서 연산이 진행이 되고 행렬과 벡터의 연산에서 벡터를 좌측에 놓는 벡터 \* 행렬 형태를 취하기 때문에 HLSL의 행렬형은 "col\_major"가 됩니다. 또한 저 수준 언어와 다르게 미리 행렬 변수 값을 Transpose를 할 필요가 없습니다.

행렬의 성분 접근은 행과 열의 인덱스를 이용한 방법 "\_m+인덱스 숫자"를 이용한 방법, [] 연산자를 이용한 방법 등 3가지가 있습니다.

```
._m{row}{col}    → ._m00 ~ ._m33
._{row}{col}     → ._11 ~ ._44
.[{row}][{col}] → mtTM[0][0] ~ mtTM[3][3]
```

행렬은 C언어와 유사하게 1차원 배열 또는 2차원 배열을 이용하거나 성분 접근으로 초기화 합니다.

```
float2x2 v = {1,2,3,4}; → v._11=1, v._12=2, v._21=3, v._22 = 4;
float2x2 v = float2x2({1,2}, {3,4});
```

행렬도 swizzling<sup>o)</sup> 가능하며 \_m{row}{col}과 \_{row}{col}을 혼용하지 않으면 됩니다. 예를 들어 v.\_m00\_11 과 같은 swizzling은 허용이 안됩니다.

복합 형(Complex Type)은 구조체(Struct), 사용자 정의(User Define), 텍스처 객체, 샘플러 객체), Shader 객체 등이 있습니다.

구조체 형은 C언어의 구조체(struct)와 유사하며 데이터의 입력(레지스터) 또는 출력(레지스터)을 지정할 때 사용됩니다.

```
struct Svertex
{
    float3 position;
    float3 normal;
};
```

struct 형 변수의 초기화는 전체를 0으로 초기할 때는 상수 "0"을 해당 구조체의 타입으로 캐스팅 합니다. 개별적인 데이터 초기화는 {}를 사용합니다. 성분 접근은 "." 연산자를 사용합니다.

초기화: struct Svertex v={ {1, 2, 3}, {4, 5, 6}};

0으로 초기화: Svertex v = (Svertex)0;

성분 접근 - 갱신: v.normal = float3(1, 0, 1);

성분 접근 - 복사: float3 t = v.position

사용자 정의는 C언어와 동일하게 typedef 키워드를 사용합니다.

```
typedef vector<float, 3> point;
point v;
```

텍스처 객체는 texture 키워드 사용하며 다음은 가장 기본적인 텍스처 객체 생성입니다.

```
texture tex0;
```

때로는 쉐이더 프로그램에는 전혀 영향을 주지 않고 응용 프로그램에서 텍스처 객체의 이름 등을 참고하고자 할 때 "<>"를 사용해서 주해(annotation)를 지정합니다.

```
texture tex0 <string name ="MyTexture.bmp">;
```

샘플러 객체는 텍스처에서 픽셀을 가져오는 샘플링을 담당하는 객체입니다. 샘플러 타입은 1차원 텍스처에 대한 sampler1D, 2차원 텍스처에 대한 sampler2D, 3차원 텍스처에 대한 sampler3D, 입방체(Cube) 텍스처에 대한 samplerCUBE 등이 있습니다.

단순한 형태의 샘플러 타입 객체를 생성은 sampler 키워드를 사용합니다.

```
sampler SampDif;
```

샘플링에서 필터링(Filtering), 어드레싱(Addressing)은 항상 같이 설정해야 합니다. 샘플러 타입의 객체는 sampler state를 사용해서 이들을 지정할 수 있습니다.

```
sampler3D MySampler = sampler_state
{
    MinFilter = LINEAR;
    MaxFilter = LINEAR;
    MipFilter = LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

하나의 샘플러를 가지고 여러 텍스처에 적용하는 것도 좋지만 대부분의 프로그래머들은 각

텍스처마다 샘플러 객체 하나씩 결합하는 형태로 쉐이더를 작성합니다. 다음은 샘플러 객체에 텍스처 객체를 지정하는 예입니다.

```
texture tex0;
sampler SampNor = sampler_state
{
    Texture = <tex0>;
    MinFilter = LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
};
```

정점 쉐이더, 픽셀 쉐이더를 통합한 ID3DXEffect 쉐이더 객체는 저 수준 혹은 고 수준 쉐이더 언어를 전부 사용할 수 있습니다. 따라서 각 경우에 맞게 컴파일 옵션을 설정해야 하는데 고 수준 쉐이더 언어의 컴파일을 저 수준 Assemble 컴파일 할 때 버전을 지정하거나 C언어의 inline assembly와 같이 HLSL 내부에서 저 수준으로 쉐이더를 작성할 때 사용하는 객체가 쉐이더 객체입니다.

정점 쉐이더 객체 지정은 "VertexShader vs" 으로 시작하고 픽셀 쉐이더 객체 지정은 "PixelShader ps" 으로 시작합니다.

만약 여러분이 저 수준 언어로 쉐이더 객체를 생성한다면 asm 키워드가 필요합니다.

```
VertexShader vs = asm {"저 수준 언어"};
```

고 수준으로 작성된 쉐이더 코드는 함수로 구성되어 있으므로 compile 키워드, 대상 쉐이더 버전, 시작 함수를 지정합니다.

```
VertexShader vs = compile "쉐이더 버전" "시작 함수()";
```

다음은 픽셀 쉐이더 객체에 대한 저 수준, 고 수준 생성에 대한 예입니다.

```
PixelShader ps = asm
{
    ps_2_0
    ...
}
```

```
PixelShader ps = compile ps_2_0 PixelProc(); //PixelProc()함수를 ps_2_0으로 컴파일
```

주해(Annotation)는 ID3DXEffect 객체를 사용할 때 파라미터에 사용자 정보를 추가하는 용도로 사용되며 응용프로그램에서 Lookup에 대한 질의 함수를 통해서 정보를 가져옵니다. HLSL 자체에는 영향을 주지 않습니다.

사용 방법은 "<", ">" 안에 데이터 타입, 변수 이름, 등호, 데이터 값 등을 ";"으로 구분해서 작성합니다.

```
texture t< string name="MyTexture.bmp";>;
```

```
technique MyTech
<
    int MyInt = 12;
    string InputArray = "MyLookup";
>
```

이렇게 HLSL에 작성된 주해는 응용 프로그램에서 질의 함수를 가지고 정보를 가져올 수 있습니다.

```
LPCSTR sName;
hAnnotation = m_pEffect->GetAnnotationByName(hTech, "InputArray");
m_pEffect->GetString( hAnnotation, &sName);
// 주해에서 InputArray 변수의 이름인 MyLookup을 가져옴
```

HLSL의 `string` 타입은 HLSL에 영향을 주지 않고 질의(Query)용도로 응용 프로그램에서 이용할 수 있도록 만든 자료 형입니다.

### 4.3 기억 장소(Storage Class)

Storage Class는 변수의 수식어로 변수의 수명(Life Time)과 접근에 대한 범위(Scope)를 결정하며 `static`, `extern`, `uniform`, `shard` 등이 있습니다.

정적 변수 `static`은 C언어의 `static`과 유사하며 전역 변수에 설정이 되면 응용 프로그램에서 접근 불가능하고 HLSL 코드 내에서만 유효 합니다. 또한 지역 변수에 설정이 되면 함수가 종료 되어도 값이 유지가 됩니다. 이것은 저 수준에서 쉐이더 내부에서 상수 레지스터를 설정할 것과 비슷합니다.

변수가 전역에 있고 어떤 수식어도 안 붙이면 외부 변수 `extern`이 됩니다. `extern` 변수는 응용프로그램에서 값을 변경할 수 있습니다. 여러분이 변환 행렬, 빛의 방향, 색상 등을 쉐이더에 전달할 경우 변수는 `extern`이 되어야 합니다.

extern 변수는 상수 테이블을 이용해서 값을 변경합니다. 상수 테이블은 HLSL 코드를 컴파일 할 때 사용한 함수 D3DXCompileShader()의 인수 리스트에서 상수 테이블인 ID3DXConstantTable\*의 주소를 얻을 수 있습니다.

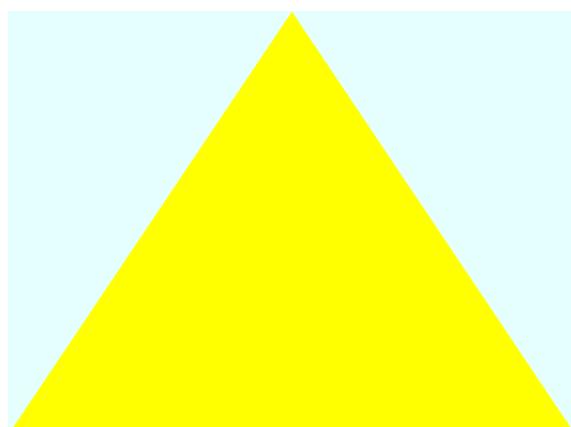
이 상수 테이블은 쉐이더의 기본 데이터 형에 대한 변수 설정 함수 SetVector(), SetMatrix(), SetInt(), SetFloat() 등이 있으며 변수를 변경하기 위해 변수 이름을 사용하는 방법과 변수 이름의 핸들을 이용한 방법 2가지가 있습니다.

```
// HLSL 코드
float4 Dif = float4(1,1,0,1);

// 변수 이름을 이용한 방법
D3DXCOLOR      color(1,0,1,1);
m_pTbl->SetVector(m_pDev, "Dif", (D3DXVECTOR4*)&color);
```

핸들을 사용하면 데이터 접근 속도에 이득이 있다고 합니다. 변수의 핸들을 얻는 것은 GetConstantByName() 함수에 이름을 전달합니다.

```
D3DXHANDLE      m_hDif; // Diffuse Constant Handle
...
// 핸들을 이용한 방법
m_hDif = m_pTbl->GetConstantByName(NULL, "Dif");
...
m_pTbl->SetVector(m_pDev, m_hDif, (D3DXVECTOR4*)&color);
```



<전역 변수 설정: [hs1\\_basic2\\_const.zip](#)>

정점 쉐이더와 퍽셀 쉐이더는 varying과 uniform, 두 종류의 입력 데이터를 받습니다. 정점 쉐이더

에서 varing 데이터는 정점의 위치, 법선, 색상 등과 같이 정점 스트림에서 온 데이터입니다. 이 와 대비되는 uniform 데이터는 저 수준으로 비교하면 상수 레지스터에 저장된 데이터라 할 수 있습니다.

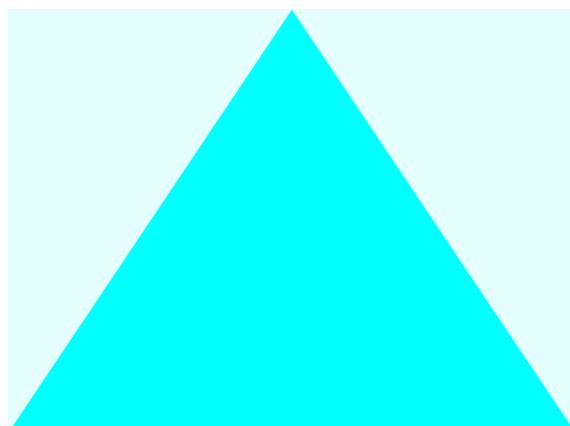
만약 여러분이 정점 처리에 대한 함수를 다음과 같이 쉐이더를 작성하면 컴파일 할 때 에러가 발생합니다.

```
void VtxPrc( in float3 iPos: POSITION
    , out float4 oPos: POSITION
    , out float4 oD0 : COLOR
    , float4 Dif // float4 앞에 uniform을 붙이면 에러 없어짐
)
{
    Dif = float4(0,1,1,1);
    oPos = float4(iPos, 1);
    oD0 = Dif;
}
```

이것은 VtxPrc() 함수가 정점 처리의 void 형 main 함수이기 때문에 입/출력을 명시적으로 지정해야 하는데 Dif 변수는 입력(in) 또는 출력(out)으로 지정되지 않았기 때문입니다. 그런데 uniform을 사용해서 "uniform float4 Dif"와 같이 작성하면 쉐이더 컴파일러는 Dif 변수를 상수 레지스터에 저장된 데이터라 여기고 컴파일을 완성합니다.

uniform 사용에서 지역 변수에는 uniform 선언이 안되고, 전역 변수와 함수의 인수에 사용되는데, D3D에서 전역 변수에 대한 uniform 선언은 extern과 거의 같습니다.

참고로 OpenGL 쉐이더 GLSL의 경우 외부에서 쉐이더의 상수 값 설정을 하기 위해서 전역 변수에 uniform으로 선언하며 위치, 색상, 법선 등의 입력 값에 대한 변수에는 var ing을 이용합니다.



<Uniform 예: [hs1\\_basic3\\_uniform.zip](#)>

## 4.4 Semantic

저 수준 쉐이더 작성에서 우리는 레지스터를 직접적으로 다루었습니다. HLSL에서 변수는 Semantic(시맨틱: 의미)을 설정할 수 있는데 Semantic은 변수의 입력과 출력을 확인 또는 지정하거나 데이터의 출처와 역할에 대한 분명한 의미를 부여하기 위해 함수, 변수, 인수 뒤에 선택적으로 붙여서 서술하는 것입니다. 종류는 Vertex Shader Semantic, Pixel Shader Semantic, Explicit Register Binding(명시적 레지스터 바인딩) 등이 있습니다.

Semantic 설정 방법은 전역 변수, 인수, 함수 등의 끝에 콜론(":")을 붙이고 "SEMANTIC 이름"을 붙입니다.

```
float4x4 m_mtWldView : WORLDVIEW;
float4 pos: POSITION;
```

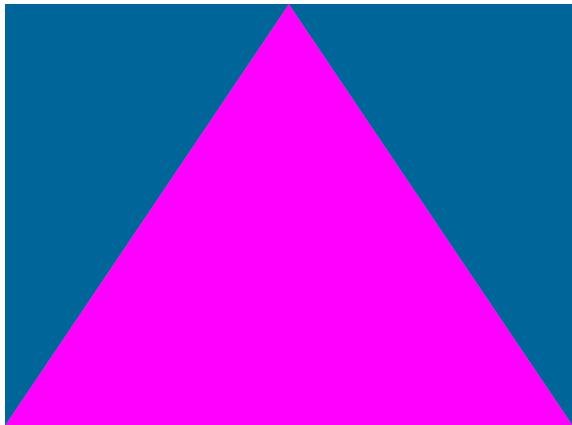
Semantic은 다음과 같이 함수의 입력 변수 또는 함수의 끝에 붙일 수 있습니다.

```
float4 MyFunc( float3 pos: POSITION) : POSITION
{
    return float4(pos, 1);
}
```

만약 이 함수가 시작 함수로 설정이 되면 입력 변수에 설정된 Semantic은 저 수준의 "dcl" 선언과 같아져 앞의 "pos" 변수는 "dcl\_position v0"와 비슷한 의미가 됩니다. 또한 함수의 뒤에 붙은 Semantic은 결과를 oPos에 복사하는 것과 같은 의미가 됩니다.

구조체 변수 다음에도 Semantic을 붙일 수 있습니다.

```
struct T
{
    float3 pos : POSITION;
    ...
};
```



<Semantic: [hs1\\_basic4\\_semantic.zip](#)>

이렇게 구조체 변수에 Semantic을 부여하고 함수의 입력 변수의 타입과, 출력의 타입으로 설정하면 프로그램 하기가 무척 편리합니다.

저 수준 쉐이더의 "dcl"과 대응되는 입력에 대한 Semantic은 다음과 같습니다.

POSITION[n]: 정점의 위치. POSITION, POSITION0, POSITION1, ...

BLENDWEIGHT[n]: 정점 블렌딩 비중 값

BLENDINDICES[n]: 정점 블렌딩 인덱스 값

NORMAL[n]: 정점 법선 벡터

PSIZE[n]: Point Size

COLOR[n]: 정점 Diffuse(COLOR, COLOR0), Specular(COLOR1)

TEXCOORD[n]: 텍스처 좌표

TANGENT[n]: 정점 접선 벡터

BINORMAL[n]: 정점 종법선 벡터

TESSFACTOR[n]: tessellation factor

정점 쉐이더의 출력 레지스터 지정하는 저 수준 명령어 oXXX와 대응되는 정점 쉐이더 출력 Semantic은 다음과 같습니다.

POSITION: 정점의 출력 위치 → oPos

PSIZE: Point Size → oPts

FOG: 정점 포그 값 → oFog

COLOR[0,1]: Diffuse, Specular → oD0, oD1

TEXCOORD[0~7]: 텍스처 좌표 → oT0~T7

D3D에서 사용자가 만든 데이터는 출력 레지스터가 없기 때문에 이 데이터를 픽셀 쉐이더로 전달하기 위한 경우 TEXCOORD 출력 시맨틱을 가장 많이 사용하며 TEXCOORD0, TEXCOORD1은 대부분 사용하

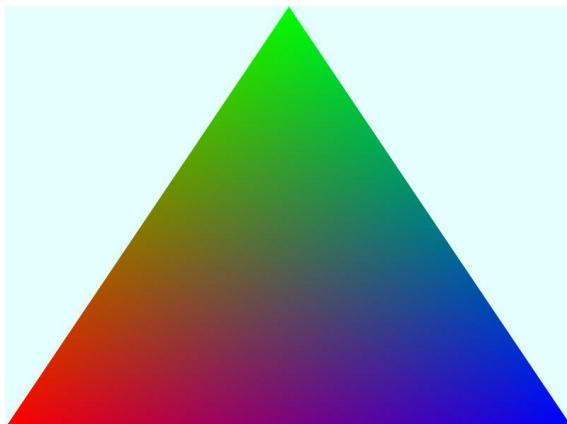
기 때문에 보통 TEXCOORD7부터 TEXCOORD6, TEXCOORD5, … 등을 이용합니다.

다음은 정점 쉐이더 Semantic을 구조체에 적용해서 사용한 예 입니다.

```
// 정점 입력에 대한 구조체
struct SvtxIn
{
    float3 Pos : POSITION;
    float3 Nor : NORMAL;
    float4 Dif : COLOR;
    float2 Tex : TEXCOORD;
};

// 정점 출력에 대한 구조체
struct SvtxOut
{
    float4 Pos : POSITION;
    float4 Dif : COLOR;
    float2 Tex : TEXCOORD0;
    float4 Nor : TEXCOORD7; // 법선 벡터는 출력 레지스터가 없으므로 TEXCOORD 이용
};

...
SvtxOut VtxPrc(SvtxIn pVtx)
{
    SvtxOut Out=(SvtxOut)0;
    // 쉐이더 연산
    ...
    return Out;
}
```



<Struct: [hs1\\_basic5\\_struct.zip](#)>

픽셀 쉐이더의 Semantic 문법은 정점 쉐이더 Semantic과 동일하게 데이터의 출처에 대한 식별을 위해 사용되며 시맨틱 위치는 구조체 멤버 뒤, 함수의 인수 뒤, 함수 뒤 등에 붙여서 사용합니다. 이 픽셀 쉐이더 Semantic은 Pixel Shader Input Semantic과 Pixel Shader Output Semantic 두 종류가 있습니다.

픽셀 쉐이더 입력 Semantic은 픽셀 쉐이더의 입력 레지스터 지정하고 Sampler 객체는 따로 지정해서 사용합니다.

픽셀 쉐이더 입력 Semantic은 COLOR, TEXCOORD 두 종류가 있습니다.

COLOR, COLOR[0, 1]: 정점 처리 과정에서 만들어진 Diffuse(0)와 Specular(1)

TEXCOORD, TEXCOORD[0~7]: 텍스처 좌표

픽셀 쉐이더 출력 Semantic은 픽셀 쉐이더의 출력 레지스터 지정하는 것으로 저 수준 명령어의 oXXXX와 대응됩니다.

COLOR[n] 색상 → oC[n], COLOR = COLOR0

TEXCOORD[n]: → 텍스처 좌표

DEPTH[n]: → oDepth[n], DEPTH = DEPTH0

Explicit Register Binding은 레지스터 이름을 명시적으로 지정하는 것으로 저 수준과 혼용해 사용하거나 아니면 고정 기능 파이프라인의 렌더링 상태 머신에 설정된 값을 이용하고자 할 때 사용합니다.

명시적 레지스터 바인딩은 다음과 같이 Register 키워드와 저 수준 레지스터 이름을 사용합니다.

타입 + 변수 + ":" + register( "저 수준 레지스터 이름");

예를 들어 고정기능 파이프라인의 다중 처리(Multi-Texturing)의 샘플러 0번에 대해서 쉐이더 코드를 다음과 같이 작성할 수 있습니다.

```
sampler SmpDif : register(s0); // Sampler 객체 SmpDif를 register s0에 바인딩
```

이렇게 되면 디바이스의 SetTexture(0, pTex); 에 바인딩 된 pTex 텍스처를 쉐이더에서 사용할 수 있습니다.

때로는 상수 레지스터를 설정할 수도 있습니다.

```
float4x4 mtWorld : register(c0); // 전역 변수 mtWorld를 상수 레지스터 c0에 바인딩
float4 vcLight : register(c10); // 전역 변수 vcLight를 상수 레지스터 c10에 바인딩
```

상수 테이블을 통해서 전역 변수 값을 설정하지만 이와 같이 레지스터를 명시적으로 바인딩 하면 상수 테이블 대신 디바이스의 SetVertexShaderConstantF() 함수로 c0, c10 레지스터 값을 변경할 수 있게 되어 결과적으로 mtWorld, vcLight를 수정한 것과 동일한 효과를 발휘합니다.

## 4.5 사용자 정의 함수

HLSL의 함수는 C언어의 함수와 거의 같으며 기본적인 형태는 다음과 같습니다.

```
"return type" "Function_Name"({"argument type" "argument name"}) { : Semantic}
{
    // function body
    return ...;
}
```

만약 함수가 정점 처리 또는 퍽셀 처리의 main 함수 경우에는 함수의 반환 형은 정점 쉐이더/픽셀 쉐이더 Semantic을 따라야 합니다.

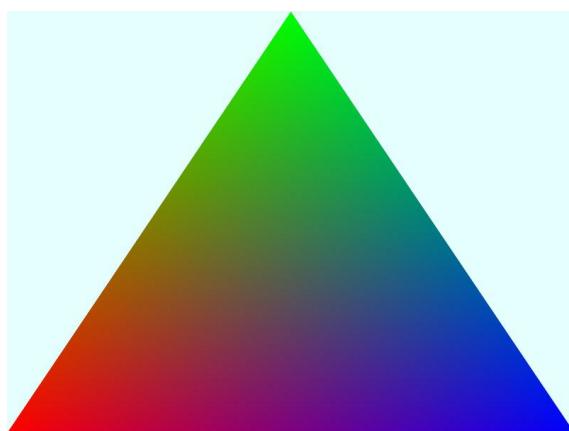
픽셀 쉐이더 main 함수 예)

```
float4 Px1Prc(float2 vTex : TEXCOORD0) : COLOR0
{
    return tex2D(DiffuseSampler, vTex);
}
```

또한 함수의 반환 형이 void 형이면 함수의 인수 형 앞에 in/out 키워드를 넣어서 입/출력을 지정할 수 있습니다.

정점 쉐이더 main 함수 예)

```
void VtxPrc( in float4 vPos : POSITION,           // 입력 레지스터 위치
              in float2 vTex : TEXCOORD0,          // 입력 레지스터 텍스처 좌표 0
              out float4 oPos : POSITION,          // 출력 레지스터 위치
              out float2 oTex : TEXCOORD0          // 출력 레지스터 텍스처 좌표 0
)
{
...
}
```



<함수 사용: [hs1\\_basic6\\_function.zip](#)>

함수의 인수 값 변동이 쉐이더 내부에서 불허할 경우 `uniform` 키워드 이용합니다.

## 4.6 HLSL 내장 함수, 기타 문법

여러분이 HLSL 내장 함수를 적절히 사용하려면 쉐이더 버전이 2.0이상으로 컴파일 하는 것이 좋습니다. 대부분 쉐이더의 함수는 정점 처리의 변환과 조명 효과 그리고 퍽셀 처리에 관련된 함수로 수학 함수와 샘플링 함수가 90% 이상입니다.

Math 함수:

- 삼각 함수: sin, cos, tan, acos, asin, cot
- 지수 함수: exp: 자연대수 e의 승수, exp2(): 2의 승수, pow(x,y)= $x^y$
- 로그 함수: log, log2, log10
- 벡터 함수: dot(), cross(), distance(), length(), len(), sqrt(), rsqrt(),
 normalize(), reflect(), refract()

- 행렬 함수: determinant(), transpose()
- 산술 함수: abs(), ceil(), floor(), round(), fmod(), lerp(), saturate()

샘플링 함수:

- tex{n}D(s,t): n차원 샘플러 s로 t 좌표에 대한 색상 추출. tex1D, tex2D, texCUBE(s,t)
- tex{n}Dproj(s,t): 정점의 깊이(z,w)를 저장한 텍스처에 대한 n차원 투영 샘플링.  
t = 4D임에 주의 샘플링 전에 t는 t.w로 나누어짐
- tex{n}Dbias(s,t): 보정된 n차원 텍스처에 대한 Sampling

HLSL의 Keyword 중에 **Technique** 있습니다. 이것은 정점 쉐이더와 픽셀 쉐이더를 통합한 ID3DXEffect 객체를 사용할 때 이용되는 키워드로 렌더링의 단위인 pass와 pass 안에 렌더링 상태 머신의 옵션 지정, 정점 쉐이더, 픽셀 쉐이더 객체의 컴파일 방법 등을 지정할 수 있습니다.

```
technique T
{
    pass P0
    {
        FogEnable      = FALSE;
        AlphaBlendEnable= FALSE;
        ...
        VertexShader = compile vs_2_0 VtxPrc();
        PixelShader = compile ps_2_0 PxlPrc();
    }

    pass P1
    {
        ...
    }
}
```

## 4.7 HLSL for Vertex Processing

지금까지 HLSL의 중요한 문법들을 살펴보았습니다. 이제 정점 처리과정부터 픽셀 처리 과정을 간단한 연습을 통해서 HLSL을 익혀보도록 합시다.

### 4.7.1 정점 변환, 텍스처

쉐이더 기초 강의 때부터 계속 이야기 했듯이 HLSL의 정점 처리 적용 범위는 변환과 조명입니다. 변환을 수식으로 표현하면 "행렬 \* 벡터" 또는 "벡터 \* 행렬"입니다. HLSL은 d3d와 일치하도록 쉐이더가 구성되어 있어서 "벡터 \* 행렬" 식을 사용합니다. 이를 곱셈은 내장 함수 "mul"을 이용합니다.

쉐이더에 입력되는 정점의 위치는 특별한 경우가 아니면 대부분 float3 형입니다. 그런데 파이프라인은 float4 형을 사용하므로 입력 데이터를 float4 형으로 늘리고 마지막 w=1로 합니다. 이렇게 해서 월드 변환, 뷰 변환, 정규 변환의 연산을 mul() 함수를 통해서 진행합니다.

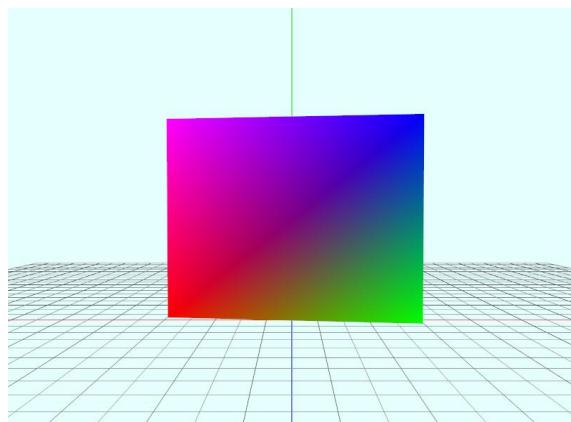
만약 뷰 변화, 정규 변화에 대해서 특별하게 처리할 내용이 없다면 외부에서 뷰 행렬과 투영 행렬을 미리 곱한 다음 쉐이더에 적용하는 것이 일반적입니다.

```

float4x4      m_mtWld;           // 월드 변환 행렬
float4x4      m_mtViewPrj;       // 뷰 * 투영 변환 행렬
...
float3        vcInput;           // 정점 데이터의 입력 위치
float4         vcOut;            // 변환된 출력 위치

vcOut = float4(vcIn, 1);        // float4형으로 늘리고 w=1로 설정
vcOut = mul(vcOut, m_mtWld);   // 월드 변환
vcOut = mul(vcOut, m_mtViewPrj); // 뷰 * 투영 변환
return vcOut

```



<정점 변환 1: [hs2\\_vtx1\\_transform1.zip](#)>

여러분은 앞으로 월드 변환, 뷰 변환, 투영 변환에 대해서 HLSL로 작성할 일이 많을 것입니다. 이들을 각각 처리해 보는 것도 아주 중요합니다.

```

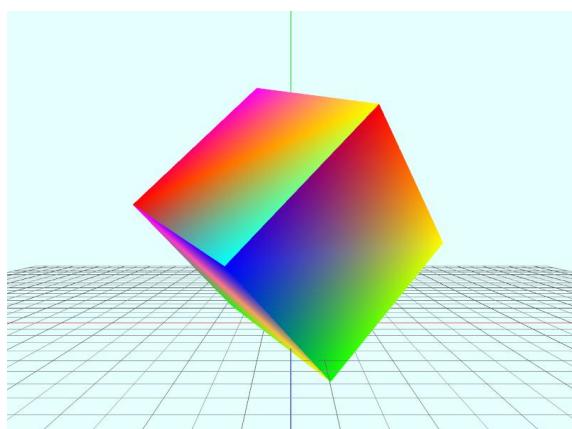
float4x4      mtWorld;          // 월드 변환 행렬
float4x4      mtView;           // 뷰 변환 행렬

```

```

float4x4      mtProj;           // 정규 변환 행렬
...
float3        vcInput;          // 정점 데이터의 입력 위치
float4        vcOut;            // 변환된 출력 위치
...
vcOut = float4(vcInput, 1);
vcOut = mul(vcOut, mtWorld);
vcOut = mul(vcOut, mtView);
vcOut = mul(vcOut, mtProj);
return vcOut;

```



<정점 변환2: [hs2\\_vtx1\\_transform2.zip](#)>

정점 처리에 대한 쉐이더 함수를 작성할 때 색상, 텍스처 좌표 등도 작성해야 합니다. 보통 텍스처의 좌표는 그림자, 환경 매핑 등과 같이 어느 특정한 대상에 대한 변환을 제외한다면 있는 그대로 출력하는 것이 대부분이며 이 경우에 대한 출력 구조체는 간단하게 만들 수 있습니다.

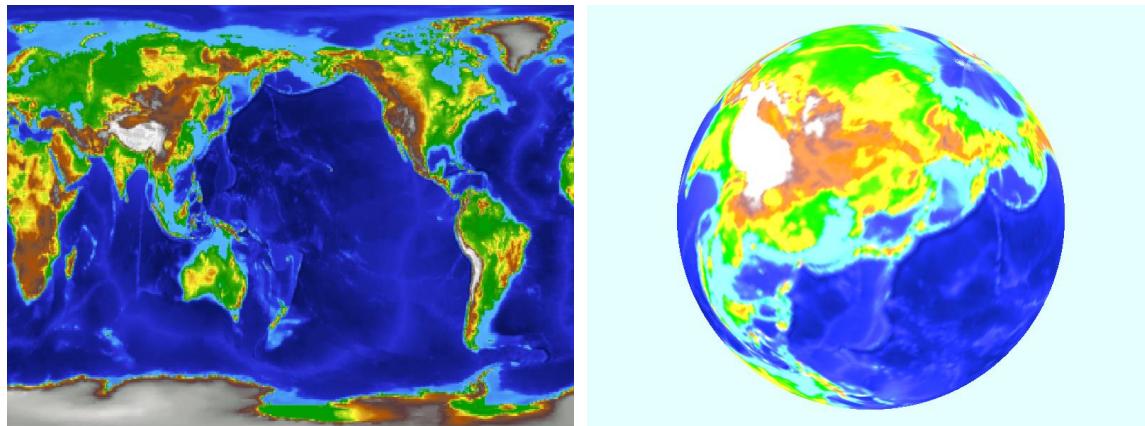
```

struct SvsOut
{
    float4 Pos : POSITION;           // 출력 위치
    float4 Dif : COLOR0;            // 출력 Diffuse
    float2 Tx0 : TEXCOORD0;          // 출력 텍스처 좌표
};

...
SvsOut VtxPrc( float3 Pos : POSITION // 입력 위치
                , float4 Dif : COLOR0 // 입력 Diffuse
                , float4 Tx0 : TEXCOORD0 // 입력 텍스처 좌표
)

```

```
{
    SvsOut Out = (SvsOut)0;
    ...
    Out.Dif = Dif;
    Out.Tx0 = Tx0;
    return Out;
}
```



<텍스처 좌표와 정점 Diffuse 출력: [hs2\\_vtx2\\_tex.zip](#), [hs2\\_vtx2\\_tex\\_earth.zip](#)>

#### 4.7.2 Diffuse color, Fog

정점의 위치 이외에 색상, 텍스처 좌표, 안개 효과 등을 하나의 함수에서 출력하기 위해서 구조체 사용이 필요합니다. 간단하게 정점의 색상을 출력하기 위한 구조체와 함수는 다음과 같이 작성할 수 있습니다.

```
struct SvsOut
{
    float4 Pos : POSITION; // 출력 위치
    float4 Dif : COLOR0; // 출력 Diffuse
};

SvsOut VtxProc( float3 Pos : POSITION, float4 Dif : COLOR)
{
    SvsOut Out = (SvsOut)0;
    // 정점 변환
    ...
}
```

```
// 색상 출력
Out.Dif = Dif;
return Out;
}
```

정점의 색상을 출력할 때 주의할 것은 색상에 대한 입력 값이 float4 형입니다. 보통 D3D는 색상을 32비트로 취급하지만 쉐이더는 r, g, b, a에 대해서 float형으로 처리하기 때문에 float4형으로 선언해야 합니다. 색상 처리를 float형으로 정의하고 색상의 범위를 [0, 1]로 정하면 색상의 가산연산에 대해서는 덧셈을, 감산 연산은 곱셈을 적용할 수가 있습니다.



<정점 Diffuse 출력: [hs2\\_vtx3\\_diffuse.zip](#)>

우리는 이전에 저 수준 정점 쉐이더에서 선형 포그(Linear Fog)를 직접 계산해서 레지스터 oFog에 복사하지 않고 정점의 색상 출력 oD0에 출력해 보았습니다. 코드의 가독성을 위해서 전 처리문을 이용해 레지스터의 이름을 우리가 생각한 이름으로 만들었지만 여전히 저 수준 명령문은 익숙해지기가 어렵습니다. HLSL을 사용하면 여러분은 선형 포그, 지수 포그, 높이 포그 등을 쉽게 구현할 수 있습니다.

선형 변화에 대한 안개 효과에 대한 공식을 출력 색상으로 지정하는 방법은 다음과 같습니다.

$$\text{Fog Factor} = \text{뷰 변환 후 정점의 } z \text{ 값} / (\text{포그 끝 값} - \text{포그 시작 값})$$

$$\text{출력 Diffuse 색상} = \text{Fog 색상} * \text{Fog Factor} + \text{정점 Diffuse} * (1 - \text{Fog Factor})$$

이 공식들을 쉐이더로 바꾸는 일은 그리 큰 어려움은 없습니다.

```
// 안개 효과 변수
float4 m_FogColor;      // Fog 색상
float   m_FogEnd;        // Fog 끝 값
float   m_FogBgn;        // Fog 시작 값
```

...

```

SvsOut VtxProc( float3 Pos : POSITION, float4 Dif : COLOR)
{
...
    float FogFactor;
    // 정점의 변환: 월드, 뷰
    vcOut = float4(vcInput, 1);
    vcOut = mul(vcOut, m_mtWorld);
    vcOut = mul(vcOut, m_mtView);

    // 정점의 뷰 변환 과정의 z/(Fog 끝 값 - Fog 시작 값)을 Fog Factor로 저장
    FogFactor = vcOut.z/(m_FogEnd - m_FogBgn);

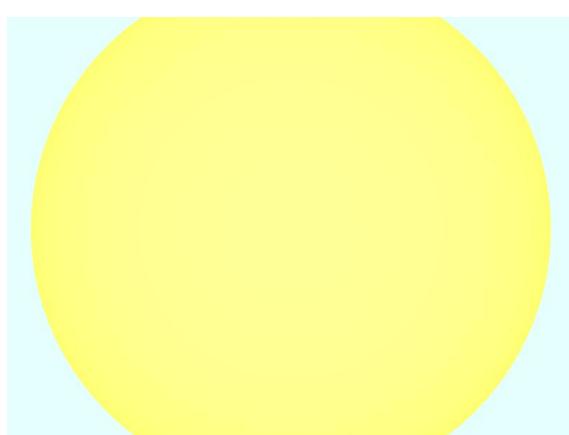
    // 정점의 정규 변환
    vcOut = mul(vcOut, m_mtProj);

    // 출력 Diffuse를 정점 Diffuse와 포그 색상과 혼합
    float4 Fog = m_FogColor * FogFactor + In.Dif * (1-FogFactor);

    // 혼합 값을 출력 색상으로 지정
    Out.Dif = Fog;

    return Out;
}

```

<안개 효과: [hs2\\_vtx3\\_fog.zip](#)>

앞의 Fog Factor는 선형적으로 안개가 변화되기 때문에 Linear Fog입니다. 쉐이더의 내장 함수

`exp()` 함수와 `pow()`를 사용해서 D3DFOG\_EXP, \_EXP2를 만들 수 있습니다.

Fog Factor EXP =  $1/\exp(\text{뷰 변환 후 정점의 } z \text{ 값} * \text{Fog Density})$

Fog Factor EXP2 =  $1/\exp(\text{pow(뷰 변환 후 정점의 } z \text{ 값} * \text{Fog Density}, 2))$

지금은 정점의 Diffuse 색상에 안개 효과를 적용하고 있어서 텍스처가 적용되면 제대로 표현하지 못합니다. 따라서 제대로 된 안개 표현은 퍽셀 쉐이더에서 처리하는 것이 가장 좋은 방법이며 쉐이더 Effect에서 구현해 보도록 하겠습니다.

#### 4.7.3 Lambert Lighting

저 수준보다 HLSL을 이용하는 장점 중에 하나가 내장된 수학 함수들이 많고 이들을 적절하게 이용할 수 있다는 것입니다. 여러분이 램버트 확산 또는 풍 반사의 세기(Intensity)를 구하기 위해서 저 수준으로 작성하게 되면 매번 모든 처리 과정을 작성해야 합니다. 물론 쉐이더 코드의 길이가 적어서 copy paste가 익숙해지면 별로 문제될 것이 없다고 생각하는 분들도 있지만 같은 내용이면 D3D에서 지원되는 것을 사용하라고 대부분 GPU를 만드는 회사들이 권장하고 있는데 우리는 D3D의 쉐이더가 지원하는 함수를 이용해서 분산 조명(램버트 확산) 효과와 스페큘러(풍 반사) 효과를 구현해 보겠습니다.

분산 조명은 램버트 확산에 기초를 두고 있으며 반사되는 빛의 세기(Intensity)를 빛의 방향과 정점의 법선 벡터와의 내적을 통해서 구합니다.

반사 밝기 = `Dot(N, L)`

이 공식을 그대로 적용하게 되면 최종 색상의 범위는 -1.0 ~ +1.0 가 됩니다. 간단하게 음수 값을 제거하기 위해서 `saturate`를 적용할 수 있습니다.

반사 밝기 = `saturate(Dot(N, L))`

이 공식은 HLSL의 `dot()`, `saturate()`함수를 사용해서 거의 그대로 바꿀 수 있습니다.

```
float3 m_vcLgt;           // 빛의 방향 벡터
...
VtxProc(...)
{
    float3 vcNor;          // 정점의 법선 벡터
```

```

...
float4 ReflectIntensity = saturate( dot(vcNor, m_vcLgt) );
...

```

만약 렌더링 오브젝트가 회전 변환을 한다면 정점의 법선 벡터는 내적을 구하는 dot() 함수에 적용되기 전에 회전 변환해야 합니다.

```
float3x3 m_mtRot; // 회전 행렬
```

```
...
```

```
vcNor = mul(vcNor, m_mtRot);
```

```
...
```

saturate()함수로 [0,1] 범위 값만 갖게 하는 것이 가장 간단하지만 현실 세계에서 빛은 공기 때문에 산란이 생기고 이로 인해서 90도 넘어도 약간의 반사가 있습니다. 이것을 물리적으로 구현하는 것이 가장 좋지만 다음과 같이 간단하게 처리할 수도 있습니다.

```
반사 밝기 = (Dot(N, L) + 1) /2
```

이 공식에 의해서 반사의 세기는 이전과 동일하게 [0, 1] 범위를 갖지만 법선 벡터와 빛의 방향 벡터의 각도가 90도 넘어도 반사 효과가 만들어 집니다. 이 공식을 일반화 시키면 다음과 같습니다.

```
반사 밝기 = saturate ( a * Dot(N, L) + b )
```

이것을 HLSL로 바꾸는 것은 어렵지 않습니다.

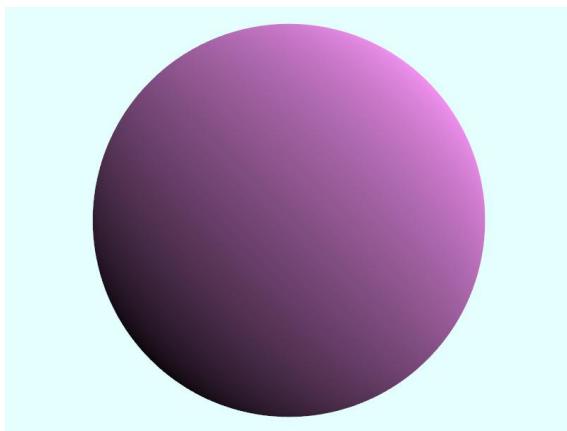
```

static float m_Sat_A = 0.5f; // Saturation Flag A
static float m_Sat_B = 0.5f; // Saturation Flag B

float3 m_vcLgt; // 빛의 방향 벡터
...
VtxProc(...)
{
    float3 vcNor = In.Nor; // 정점의 법선 벡터
    ...
    float4 ReflectIntensity = saturate(m_Sat_A * dot(vcNor, m_vcLgt) + m_Sat_B);
}

```

외부에서 Saturation 변수의 값을 바꾸고 싶으면 static 키워드를 해제하면 됩니다.



<Lambert: [hs2\\_vtx4\\_lgt\\_diffuse.zip](#)>

#### 4.7.4 스페큘러 조명

퐁 반사를 사용한 스페큘러 반사 세기는 정점의 법선 벡터(N)에 의해 반사되는 광원의 반사 벡터(R), 정점에서 바라보는 카메라에 대한 시선 벡터(E)의 내적의 결과에 하이라이트(Sharpness) 세기를 엑스포너셜(Power)으로 구합니다. 여러분은 먼저 반사 벡터(R: Reflection vector)를 정점의 법선 벡터(N)와 빛의 방향 벡터(L)을 사용해서 먼저 구하고 반사의 세기를 결정합니다.

```
R = 2 * dot(N, L) * N - L
퐁 반사 밝기 = power(Dot(R, E), Sharpness)
```

반사 벡터를 구해주는 내장 함수는 reflect()함수이며, 이 함수를 사용할 때는 빛의 방향을 반대 방향으로 설정합니다.

```
reflect(L, N) = L - 2 * dot(L, N) * N
반사 벡터 R = reflect(-L, N)
```

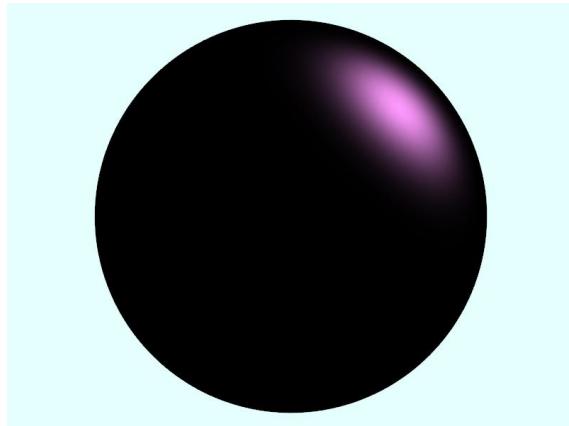
엑스포너셜(Power)을 구하는 HLSL 함수는 pow()입니다. 이를 내장 함수 등을 이용해서 퐁 반사의 세기를 HLSL로 간단하게 작성할 수 있습니다.

```
float3 m_vcLgt; // 빛의 방향 벡터
float3 vcNor; // 정점의 법선 벡터
float3 vcEye; // 정점에서 카메라 위치를 바라본 시선 방향 벡터
...
float3 vcR = reflect(-m_vcLgt, vcNor);
```

```
float4 Phong = pow( dot(vcR, vcEye), m_fSharp);
```

그런데 렌더링 오브젝트는 월드 변환이 필요하므로 반사의 세기를 구하기 전에 정점에서 카메라의 위치를 바라본 시선 방향 벡터와 반사벡터에 영향을 주는 정점의 위치와 법선 벡터를 각각 월드 변환과 회전 변환을 적용해야 하고 카메라의 위치를 외부에서 받아와서 시선 방향 벡터를 구한다음 반사의 세기를 구합니다.

```
float4x3 m_mtWld;           // 월드 행렬
float3x3 m_mtRot;          // 회전 행렬
float3 m_vcCam;            // 카메라 위치
float3 m_vcLgt;             // 빛의 방향 벡터
...
VtxProc(...)
{
    float3 vcPos = mul(float4(In.Pos, 1), m_mtWld); // 위치의 월드 변환
    float3 vcNor = mul(In.Nor, m_mtRot);           // 법선의 회전 변환
    float3 vcEye = normalize(m_vcCam - vcPos);      // 시선 벡터의 정규화
    float3 vcRfc = reflect(-m_vcLgt, vcNor);        // 반사 벡터
    float4 Phong = pow( dot(vcRfc, vcEye), m_fSharp); // 풍 반사 세기
```



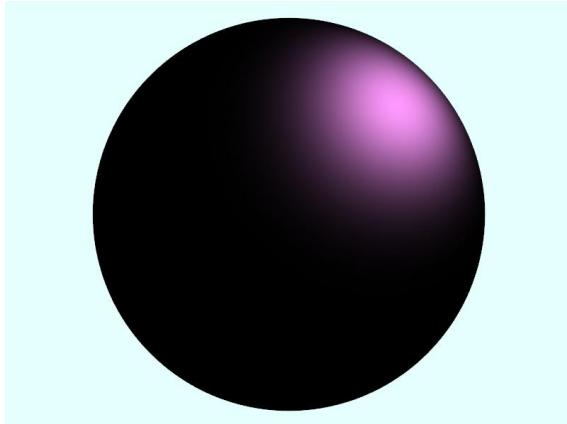
<퐁 반사: [hs2\\_vtx4\\_lgt\\_phong.zip](#)>

Blinn-Phong 반사의 세기는 풍 반사의 반사 벡터와 시선 벡터를 사용하지 않고 Half 벡터와 정점의 법선 벡터를 이용합니다.

Half 벡터 = normalize(E + L)

Blinn-Phong 반사 세기 = Dot(N, H)^Sharpness

```
VtxProc(...)  
{  
...  
    float3 vcEye = normalize(m_vcCam - vcPos);      // 시선 벡터의 정규화  
    float3 vcHlf = normalize(vcEye + m_vcLgt);      // Half 벡터  
    float4 Blinn = pow( dot(vcNor, vcHlf), m_fSharp); // Blinn-Phong 반사 세기
```



<블린-퐁 반사: [hs2\\_vtx4\\_lgt\\_blinn.zip](#)>

## 4.8 HLSL for Pixel Processing

### 4.8.1 Simple HLSL for Pixel Shader

프로그래밍 가능한 퍽셀 파이프라인은 퍽셀의 샘플링(Sampling)과 다중 텍스처 처리(Multi-Texturing)입니다. 퍽셀 파이프라인에 입력되는 데이터는 정점 처리 과정의 Rasterizing을 통해서 만들어진 퍽셀, 텍스처 좌표, 그리고 텍스처입니다.

퍽셀 처리의 결과는 색상이기 때문에 HLSL 함수를 작성하면 출력은 float4형으로 정하고 Semantic은 COLOR로 합니다. 간단하게 정점 처리 과정에서 만들어진 색상을 그대로 출력하는 HLSL 함수를 다음과 같이 작성할 수 있습니다.

```
float4 Px1Prc(float4 iDif: COLOR) : COLOR  
{  
    return iDif;  
}
```

void형 함수를 사용하는 경우라면 in/out 키워드를 이용해서 다음과 같이 작성합니다.

```

void PxlPrc(    in float4 iDif : COLOR0      // From Vertex Processing
                ,        out float4 oDif: COLOR0      // Output oC0
)
{
    oDif = iDif;
}

```

정점 처리의 HLSL과 마찬가지로 이를 함수 또한 D3DXCompileShader() 함수를 사용해서 컴파일을 해야 합니다.

```

LPD3DXBUFFER pShd    = NULL;
LPD3DXBUFFER pErr    = NULL;
hr = D3DXCompileShader(sHlsl, iLen
                      , NULL, NULL
                      , "PxlPrc"      // 시작 함수
                      , "ps_1_1"       // 쉐이더 버전
                      , dwFlags
                      , &pShd         // 컴파일 쉐이더
                      , &pErr         // Error
                      , &m_pTbl        // 상수 테이블
);

```

D3DXCompileShader() 함수는 HLSL로 작성한 정점 쉐이더, 픽셀 쉐이더 등을 컴파일하며 이후의 절차적 텍스처에 대한 쉐이더 또한 컴파일을 합니다.

간단히 출력하는 쉐이더 코드는 ps.1.1 으로도 충분하지만 현재 대부분의 그래픽 카드는 2.0 이상 지원이 되므로 D3DXCompileShader() 함수의 쉐이더 버전 인수에 ps\_2\_0 이상으로 설정하는 것이 좋습니다.

D3DXCompileShader() 함수는 단순하게 쉐이더만 컴파일 하므로 컴파일 결과를 가지고 픽셀 쉐이더를 생성해야 합니다.

```

LPDIRECT3DPIXELSHADER9 m_pPs;           // Pixel Shader
...
D3DXCompileShader(..., &pShd,...);
m_pDev->CreatePixelShader( (DWORD*)pShd->GetBufferPointer() , &m_pPs);

```

픽셀 쉐이더 사용, 즉, 프로그램 가능한 픽셀 처리 파이프라인 이용은 저 수준과 동일하며 사용이 끝나면 NULL 인수를 이용해서 사용 해지를 알립니다.

```
m_pDev->SetPixelShader(m_pPs);           // Pixel Shader 사용
...
m_pDev->DrawPrimitive(...);             // Rendering
m_pDev->SetPixelShader( NULL );        // Pixel Shader 해제
```



<픽셀 쉐이더 기초: [hs3\\_px11\\_basic.zip](#)>

#### 4.8.2 픽셀 처리 사용자 함수

저 수준은 서로 다른 처리 routine을 하나로 작성하는 것이 쉽지가 않았습니다. 하지만 HLSL을 사용하면 if 문과 함수로 여러 처리에 대해서 쉽게 작성할 수 있습니다. 이에 대한 예로 색상의 반전, 단색화 등을 HLSL로 작성해 봅시다.

```
int g_nPxlPrc = 2;                      // 픽셀 처리 타입

float4 PxlInverse(float4 Input)          // 색상 반전
{
    return 1 - Input;
}

float4 PxlMonotone(float4 InColor)       // 단색화
{
    float4 Out = 0.f;
    float4 d = float4(0.299, 0.587, 0.114, 0);
    float4 t = InColor;
```

```

    Out = dot(d, t);
    Out.w = 1;
    return Out;
}

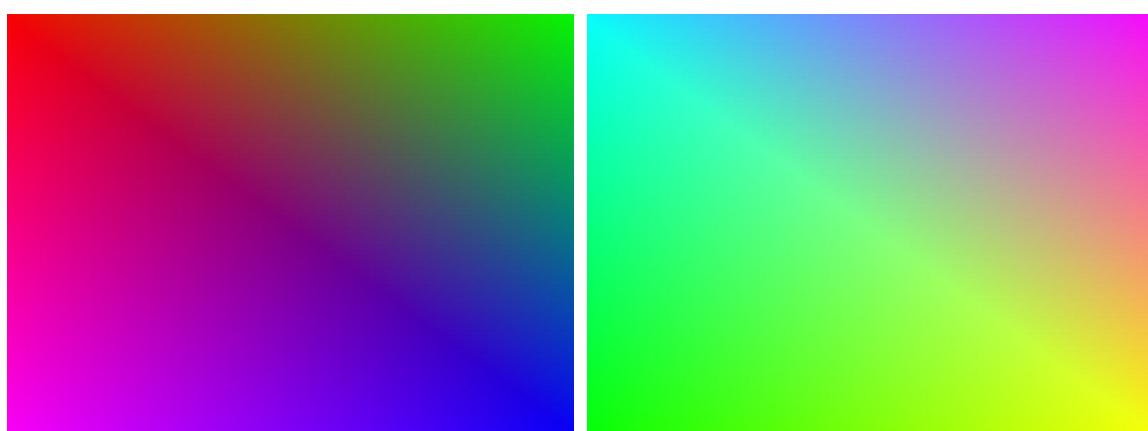
// 퍽셀 main 처리 함수
float4 PxlPrc(  in float4 iDiff : COLORO           // From Vertex Processing
) : COLORO
{
    float4 Out=float4(0,0,0,1);

    if(1== g_nPxlPrc)
        Out = PxlInverse(iDiff);
    else if(2 == g_nPxlPrc)
        Out = PxlMonotone(iDiff);
    else
        Out = iDiff;

    return Out;
}

```

색상의 반전은 1- 색상으로 처리하며 PxlInverse() 함수를 이를 구현하고 있습니다. 단색화는 r, g, b에 대해서 적절한 값을 곱해주고 더해서 이 값을 동일하게 r, g, b에 적용하는 것으로 PxlMonotone()함수가 이를 처리하고 있습니다. PxlPrc() 함수는 퍽셀 처리의 main 함수이며 색상 처리 종류(g\_nPxlrc)에 따라서 원래의 색상, 반전, 단색화를 해당 함수를 불러서 처리하고 있습니다.





<색상 반전, 단색화: [hs3\\_px12\\_inv\\_mono.zip](http://hs3_px12_inv_mono.zip)>

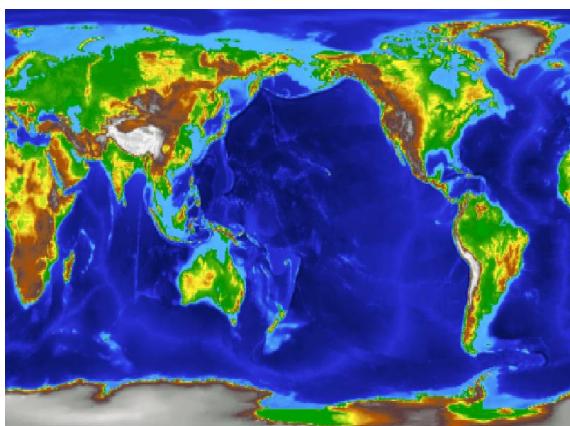
#### 4.8.3 텍스처 처리

텍스처에서 샘플링을 담당하는 샘플러를 꾹셀 쉐이더에서 다음과 같이 지정합니다.

```
sampler smpDiff;
```

이렇게 샘플러를 지정하고 나서 tex2D() 와 같은 샘플링 함수에 텍스처 좌표를 입력하면 텍스처에서 색상을 가져오게 됩니다.

```
float4 Px1Prc( in float4 Tx0 : TEXCOORD0 /* 텍스처 좌표 */) : COLOR0
{
    return tex2D(smpDiff, Tx0);      // 샘플링
}
```



<샘플러 register 선언: [hs3\\_px13\\_sampler1.zip](http://hs3_px13_sampler1.zip)>

픽셀 쉐이더의 색상 처리를 연습하는 여러 가지 방법 중의 하나가 Post Effect에서 사용되는 단색화(Monotone)와 흐림 효과(Blur Effect)를 구현해 보는 것입니다.

이전에 단색화는 색상의 r, g, b에 적당한 비중 값을 곱한 후에 이 값을 다시 r, g, b에 설정하는 것이라 했습니다. 또한 r, g, b에 대한 비중 값과 픽셀의 r, g, b를 벡터로 생각하고 내적(Dot Product)를 하면 곧 단색화의 색상이 된가도 했습니다.

단색화 값 = 단색화 비중(r, g, b), 픽셀 색상(r, g, b)

이 단색화 값에 정해진 색상을 곱하게 되면 화면에 특정한 색상으로 장면을 연출합니다. 단색화의 HLSL 구현을 위해서 이전에 저 수준으로 만들었던 예제를 HLSL로 바꾸면 다음과 같습니다.

```
sampler SampDif : register(s0);
...
float4 Out=0.0F;
float4 MonoColor ={0.5F, 1.0F, 2.0F, 1.0F};           // 단색화 색상
float4 MonoWeight={0.299F, 0.587F, 0.114F, 0.0F};      // 단색화 비중

Out = tex2D( SampDif, Tx0 );    // 채플링
Out = dot(Out, MonoWeight);     // 내적(dot)로 단색화
Out *= MonoColor;              // 단색화 색상을 곱함
...
return Out;
```



<단색화: [hs4\\_px11\\_mono.zip](#)>

흐림 효과(Blur Effect)는 인접한 픽셀에 비중 값을 곱하고 더해서 최종 색상을 만드는 것입니다.

특히 Gaussian Blur은 인접한 픽셀까지의 거리를 가지고 비중 값을  $\exp()$ 함수로 결정하며 수식으로 표현하면 다음과 같습니다.

$$\text{최종 색상} = \sum P_i * \exp(-x^2 / \Delta^2)$$

HLSL은  $\exp()$  함수를 지원합니다. 인접한 픽셀은 텍스처의 좌표를 변화시킨 후에  $\text{tex2D}()$  와 같은 샘플링 함수를 가지고 얻으며 이것을 반복적으로 `for` 문 등을 이용해서 적용합니다. 다음은 Gaussian Blur를 HLSL로 구현한 예입니다.

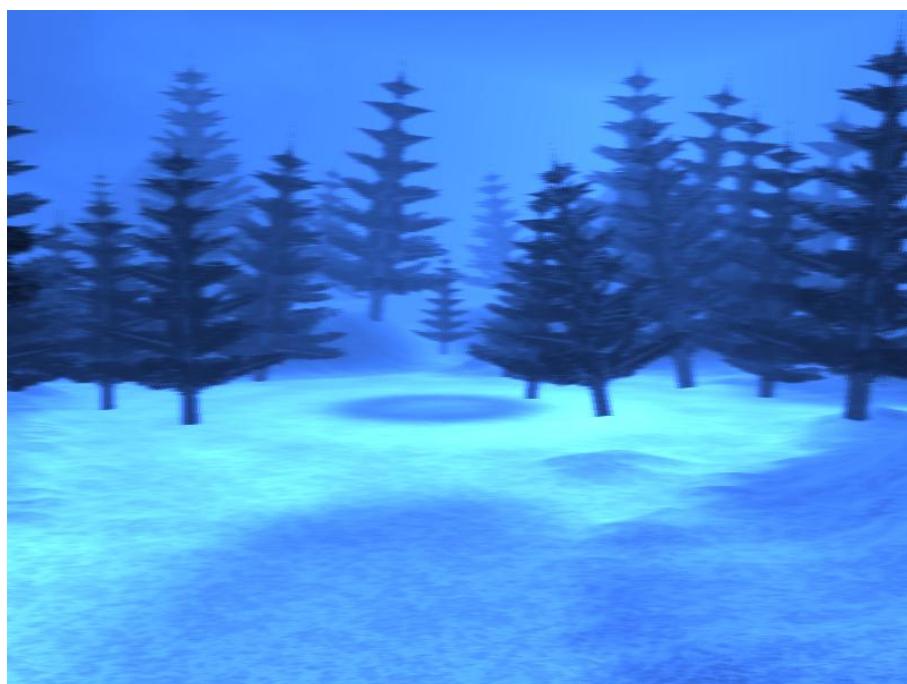
```
sampler SampDif : register(s0);
...
float4 Out=0.0F;
...
for(x, ...)
{
    float2 T = Tx0;
    T.x += (2.f * x)/1024.f;
    Out += tex2D( SampDif, T ) * exp( (-x*x)/8.f );
}
Out *= 0.24F; // 전체 명도를 낮춤
...
return Out;
```



<Blur 효과: [hs4\\_px12\\_blur.zip](#)>

단색화와 흐림 효과는 그 자체만으로 의미가 있지만 코드의 길이가 길지 않기 때문에 이 둘을 합쳐서 표현하는 것도 좋습니다.

```
sampler SampDif : register(s0);
...
float4 Out=0.0F;
...
for(int x=-4; x<=4; ++x)
{
    float2 T = Tx0;
    T.x += (2.f * x)/1024.f;           // 텍스처 좌표를 변화시킨다
    Out += tex2D( SampDif, T ) * exp( (-x*x)/8.f );
}
Out = dot(Out, MonoWeight);           // 단색화
Out *= MonoColor;                   // 단색에 적용할 색상을 곱함
...
return Out;
```



<단색화 + 흐림 효과: [hs4\\_px13\\_mono+blur.zip](#)>

흐림 효과와 단색화를 합쳐 놓으니까 더 멋진 장면을 만들어 냈습니다. 게임의 장면들은 이렇게

작은 부분을 결합해서 만드는 경우도 많이 있으니 꾸준히 연습하기 바라며 다음으로 다중 처리를 HLSL로 구현해 보도록 하겠습니다.

다중 텍스처 처리(Multi-Texturing)를 하려면 샘플러를 샘플러 레지스터에 명시적으로 선언을 해야 합니다. 이 때 register() 함수를 이용합니다.

```
sampler smp0 : register(s0);
sampler smp1 : register(s1);
sampler smp2 : register(s2);
...
```

이렇게 샘플러를 선언하면 pDevice->SetTexture(StageIndex, pTexture)와 같은 문장을 실행 할 때 Stage Index에 해당하는 레지스터에 텍스처 포인터가 자리잡고 이 텍스처를 샘플러가 샘플링 합니다.

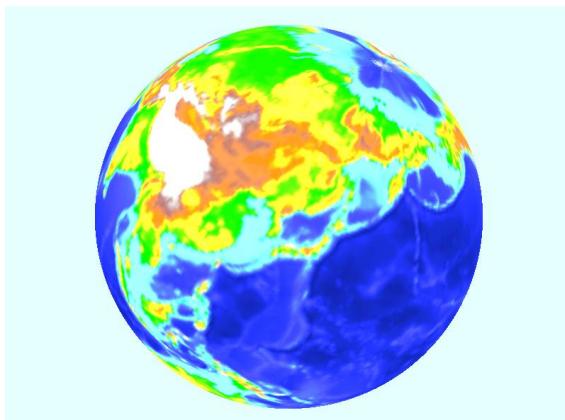
텍스처 샘플링에 대한 내장 함수는 tex1D, tex2D, tex3D, tex2Dproj, tex3Dproj, texCUBE 등이 있으며 가장 많이 사용하는 함수는 2차원 텍스처에 대한 tex2D() 함수입니다. 모든 샘플링 함수는 샘플러와 텍스처 좌표를 인수로 받습니다. 따라서 픽셀 처리의 main 함수는 정점 처리에서 만들어진 색상뿐만 아니라 텍스처 좌표도 같이 받아야 합니다.

Rasterizing 으로 만들어진 색상과 텍스처의 색상을 modulate하는 과정을 다음과 같이 작성할 수 있습니다.

```
sampler smp0 : register(s0);
float4 PxIPrc(float4 iDiff: COLOR, float2 Tex0: TEXCOORD0) : COLOR
{
    return iDiff * tex2D(smp0, Tex0);
}
```

만약 modulate2x 를 구현하고자 하면 출력의 결과에 2를 곱하면 됩니다.

```
float4 Out;
Out = tex2D(smp0, Tex0);
Out *= iDiff;
Out *= 2;
return Out;
```



<MODULATE2X: [hs3\\_px13\\_sampler2\\_earth.zip](#)>

HLSL을 사용하면 고정 기능 파이프라인의 MODULATE4X, ADD, SUBSTRACT, ADDSIGNED 등의 FLAG로 다중 처리를 지시한 내용을 간단한 산술 연산으로 구현할 수 있어서 고정 파이프라인에서 처리하는 것보다 여려모로 이점이 많습니다.

고정 기능 파이프라인에서 다중 처리에 대한 텍스처 샘플링의 상태 값을 설정하는 것은 간단하지만 D3D의 FLAG 값들을 기억하기가 어려웠습니다. 그런데 HLSL을 사용하면 각 샘플러에 대해서 필터링과 어드레싱을 설정할 수가 있습니다. 샘플러의 상태를 설정하기 위해서 sampler\_state를 사용합니다.

```
sampler smpDif0 : register(s0) = sampler_state
{
    MinFilter = POINT;           // Filtering
    MagFilter = POINT;
    MipFilter = POINT;
    AddressU = Wrap;          // Addressing
    AddressV = Wrap;
};

sampler smpDif1 : register(s1) = sampler_state
{
    MinFilter = NONE;
    MagFilter = NONE;
    MipFilter = NONE;
    AddressU = Wrap;
    AddressV = Wrap;
};
...
```



<Multi-Texturing: [hs5\\_px14\\_multi\\_tex1.zip](#)>

앞서 산술 연산으로 Multi-Texturing을 쉽게 구현할 수 있다고 했습니다. 다음의 HLSL 코드는 2개의 텍스처를 여러 상황에 대해서 간단한 산술 연산으로 구현한 예입니다.

```
sampler SampDif0 : register(s0) = sampler_state
...
sampler SampDif1 : register(s1) = sampler_state
...
int m_nMulti;
float4 Px1Prc(float4 Tx0 : TEXCOORD0) : COLOR0
{
    float4 Out= 0;
    float4 t0 = tex2D( SampDif0, Tx0 );           // Sampling m_TxDif0
    float4 t1 = tex2D( SampDif1, Tx0 );           // Sampling m_TxDif1

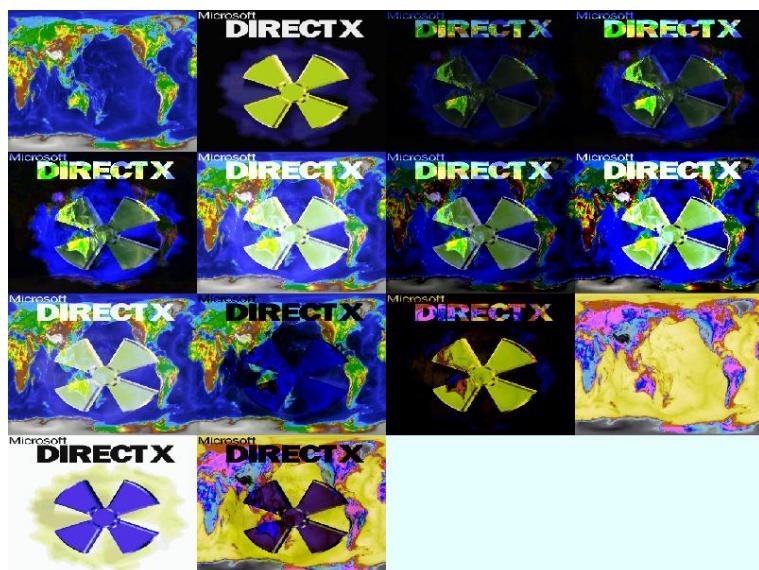
    if(0 == m_nMulti)      Out = t0;
    else if(1 == m_nMulti) Out = t1;
    else if(2 == m_nMulti) Out = t0 * t1;          // Modulate
    else if(3 == m_nMulti) Out = t0 * t1 * 2;       // Modulate 2x
    else if(4 == m_nMulti) Out = t0 * t1 * 4;       // Modulate 4x
    else if(5 == m_nMulti) Out = t0 + t1;          // Add
    else if(6 == m_nMulti) Out = t0 + t1 - .5;      // Add signed
    else if(7 == m_nMulti) Out =(t0 + t1 - .5)*2;   // Add signed
    else if(8 == m_nMulti) Out = t0 + t1 - t0*t1;   // add smooth
```

```

    else if(9 == m_nMulti) Out = t0 - t1;           // sub
    else if(10== m_nMulti) Out = t1 - t0;           // sub
    else if(11== m_nMulti) Out = 1 - t0;            // Inverse t0
    else if(12== m_nMulti) Out = 1 - t1;            // Inverse t1
    else if(13== m_nMulti) Out = 1 - (t0 + t1);     // Inverse (t0+t1)

    return Out;
}

```



<Multi-Texturing 연습: [hs5\\_px14\\_multi\\_tex2.zip](#)>

#### 4.8.4 Procedural Texture

HLSL의 흥미로운 점은 쉐이더로 작성한 퍽셀 처리를 텍스처에 저장할 수가 있습니다. 이 때 사용되는 함수가 D3DXFillTextureTX() 함수입니다. 텍스처의 퍽셀을 어떤 알고리듬(Algorithm)에 의해서 컴퓨터의 처리에 의해 만들어진 텍스처를 절차적 텍스처(Procedural Texture)라 합니다. HLSL은 Algorithm이고 이를 적용하게 되면 절차적 텍스처가 됩니다.

절차적 텍스처에 대한 쉐이더를 작성할 때 퍽셀의 좌표는 [0, 1] 범위가 되며 퍽셀 처리에 대한 main 함수의 입력 Semantic은 텍스처 좌표 TEXCOORD가 아닌 위치 POSITION으로 합니다. Position 값은 뷔포트의 영역을 [0, 1]로 바라본 값입니다. 또한 쉐이더가 퍽셀의 색상을 결정하므로 함수의 반환 값은 float4 형으로 하고 Semantic은 COLOR로 합니다.

```
float4 Tx1Prc(float2 In : POSITION) : COLOR
```

```
{
    float4 Out;
    ...
    return Out;
}
```

절차적 텍스처에 대한 쉐이더를 HLSL로 작성하게 되면 정점 쉐이더, 텍셀 쉐이더와 마찬가지로 D3DXCompileShaderFromFile() 함수를 이용해서 쉐이더를 컴파일하고 쉐이더 버전에 텍셀 버전을 입력합니다.

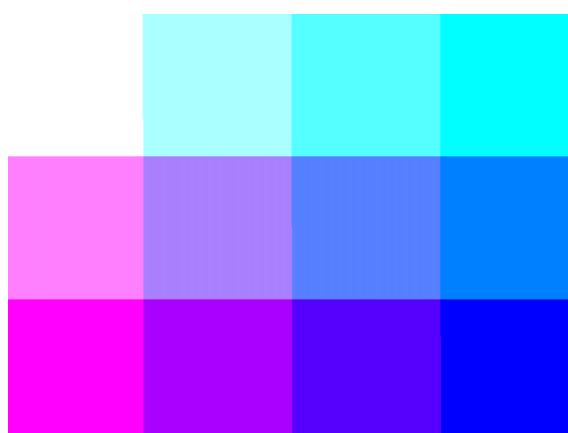
```
D3DXCompileShaderFromFile("쉐이더 파일"
    , NULL, NULL
    , "TxlPrc"      // 텍셀 main 함수
    , "tx_1_0"       // 텍셀 버전
    , dwFlags, &pShd, &pErr, NULL);
```

절차적 텍스처는 D3D Device의 CreateTexture() 함수 또는 D3DXCreateTexture() 함수를 이용해서 생성합니다.

```
D3DXCreateTexture(m_pDev, 128, 128, 1, 0, D3DFMT_UNKNOWN, D3DPOOL_MANAGED, &m_pTx);
```

텍스처의 텍셀에 대해서 텍셀 쉐이더를 적용하기 위해 D3DXFillTextureTX() 함수를 호출하면 절차적 텍스처가 완성이 됩니다.

```
D3DXFillTextureTX(m_pTx, (DWORD*)pShd->GetBufferPointer(), NULL, 0);
```

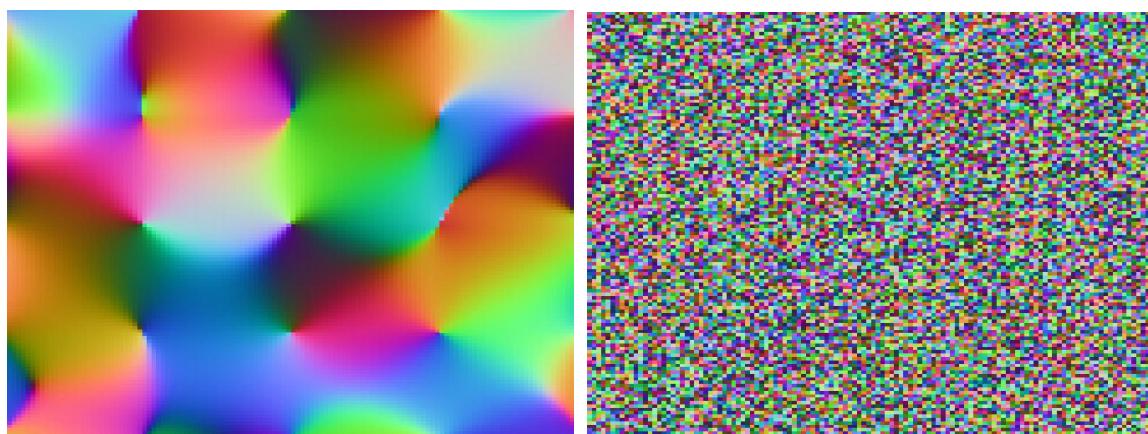


<Procedural 텍스처: [hs5\\_px14\\_procedual1.zip](#)

화면 잡음과 같은 Post Effect나 Normal Mapping 등에서는 정점의 텍스처 좌표에서 공식에 따른 위치에 텍스처 좌표를 다시 설정하고 샘플링 작업을 진행하는 경우가 있습니다. 이러한 경우에 공식을 직접 적용하는 방법도 있지만 때로는 공식의 결과 값을 텍스처에 저장해서 처리 속도를 빠르게 합니다.

[hs4\\_px14\\_procedual2.zip](#)는 Noise 텍스처를 생성하는 예제로 random 값을 만들기 위해서 noise() 함수를 사용하고 있습니다.

```
static float m_Delta = 2000.0F;
float4 TexPrc(float2 Pos : POSITION) : COLOR0
{
    float4 Out = (float4)0;
    Out.r = noise((Pos+0) * m_Delta);
    Out.g = noise((Pos+1) * m_Delta);
    Out.b = noise((Pos+2) * m_Delta);
    Out = normalize(Out);
    Out = (Out + 1) * 0.5F;
    Out.w = 1;
    return Out;
}
```



<Procedural Noise 텍스처: [hs4\\_px14\\_procedual2.zip](#) m\_Delta = 4, m\_Delta = 2000>

Post Effect 연습에서 사각형, 은행잎, 직소 등의 패턴 효과에서 일정한 영역에 대해서 같은 픽셀을 적용합니다. 영역 중심에서 벗어나는 텍스처 좌표는 중심 쪽으로 상대적으로 이동할 수 있도록 해야 하는데 이 값(Delta)을 미리 텍스처에 저장합니다.

텍스처에서 픽셀을 가져와서 새로운 좌표로 다음의 공식을 이용합니다.

`New(U, V) = Old(U, V) + Delta( r-0.5, g-0.5)`

[hs5\\_px14\\_procedual3.zip](#) 예제는 텍스처의 중앙에서 샘플링이 가능하도록 Delta 값을 저장하는 예입니다.

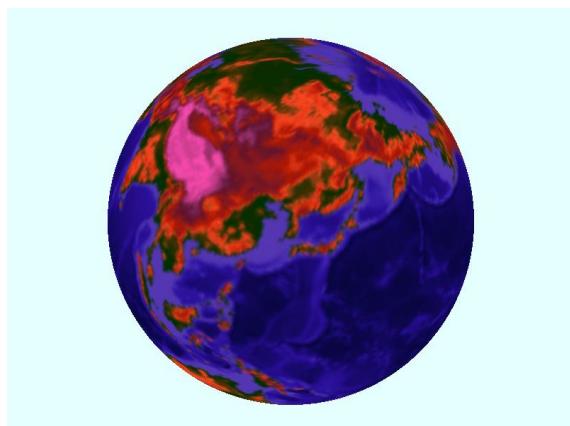


<Procedural Delta 텍스처: [hs5\\_px14\\_procedual3.zip](#)>

이 텍스처를 사용하는 예제는 Post Effect의 응행일, 직소 등을 살펴보기 바랍니다.

## 4.9 ILcEffect 만들기

저 수준 쉐이더에서는 하나의 모듈은 하나의 파일 또는 단일 문자열에서 처리했습니다. 그런데 HLSL은 정점과 픽셀 처리에 대한 main 함수들을 하나의 파일에 작성할 수 있고 D3DXCompileShaderFromFile()에서 컴파일 할 때 원하는 main 함수를 지정하면 해당 함수와 관련된 내용만 컴파일 합니다.



<정점 쉐이더, 픽셀 쉐이더를 하나의 파일에 작성한 예: [hs6\\_vtxpxl.zip](#)>

이것은 여러 가지 장점이 있습니다. D3D가 구조상 정점과 픽셀 쉐이더가 분리되어 있다 하더라도 게임에서는 이 둘을 한꺼번에 사용되는 경우가 많이 있으므로 이 둘을 하나의 모듈 안에 구현되어 있으면 편리하며 우리는 이를 위해서 고 수준 언어로 작성된 정점과 픽셀 쉐이더를 ILcEffect로 추상화 해 봅시다.

HLSL 과 저 수준 쉐이더는 언어의 형식만 다르기 때문에 저 수준에서 추상화한 방식이 거의 그대로 활용될 수 있습니다. 변화된 것은 쉐이더의 상수 설정에 대한 구현이 상수 테이블을 사용하고 정점 쉐이더와 픽셀 쉐이더의 main 함수를 지정하는 것입니다.

```
interface ILcEffect
{
    virtual INT      Create(void* =NULL, void* =NULL, void* =NULL, void* =NULL)=0;
    virtual void     Destroy()=0;
    virtual INT      Begin()=0;
    virtual INT      End()=0;

    virtual INT      SetupDecalator(DWORD dFVF)=0;
    virtual INT      SetMatrix(char* sName, D3DXMATRIX* v)=0;
    virtual INT      SetVector(char* sName, D3DXVECTOR4* v)=0;
    virtual INT      SetColor(char* sName, D3DXCOLOR* v)=0;
    virtual INT      SetFloat(char* sName, FLOAT v) =0;
};
```

ILcEffect 객체를 생성하는 함수는 다음과 같습니다.

```
int Lch1sl_CreateShader(char* sCmd, ILcEffect** pData, ...);
```

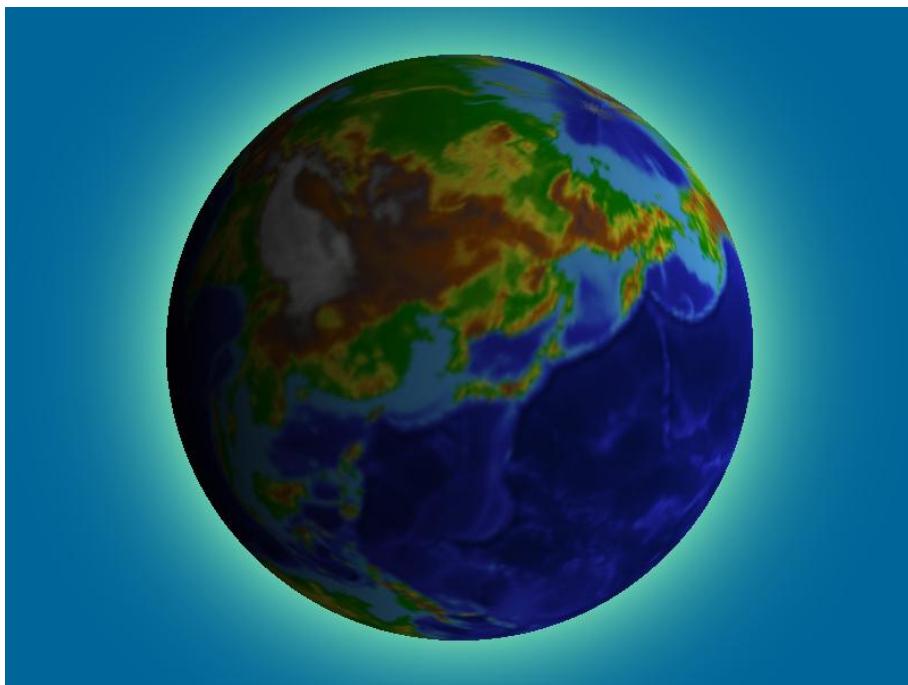
ILcEffect를 구현한 CH1slEffect은 지정된 쉐이더 함수를 컴파일하고 정점 선언자, 정점 쉐이더, 픽셀 쉐이더 객체를 생성합니다. 또한 상수 레지스터 설정을 상수 테이블로 구현합니다.

```
LPD3DXCONSTANTTABLE    m_pTbl; // 정점 쉐이더를 위한 상수 테이블
...
INT CH1slEffect::SetMatrix(char* sName, D3DXMATRIX* v)
{
    return m_pTbl->SetMatrix(m_pDev, sName, v);
}
```

```
INT CH1slEffect::SetVector(char* sName, D3DXVECTOR4* v)
```

```
{  
    return m_pTbl->SetVector(m_pDev, sName, v);  
}  
  
INT CHls1Effect::SetColor(char* sName, D3DXCOLOR* v)  
{  
    return m_pTbl->SetVector(m_pDev, sName, (D3DXVECTOR4*)v);  
}  
  
INT CHls1Effect::SetFloat(char* sName, FLOAT v)  
{  
    return m_pTbl->SetFloat(m_pDev, sName, v);  
}
```

좀 더 정교하게 작성하려면 정점 쉐이더에 대한 상수 테이블뿐만 아니라 픽셀 쉐이더의 상수 테이블도 같이 구현하는 것이 좋습니다. [hs6\\_ILcEffect.zip](#)는 ILcEffect를 구현하고 Glow 효과를 보여주고 있습니다.



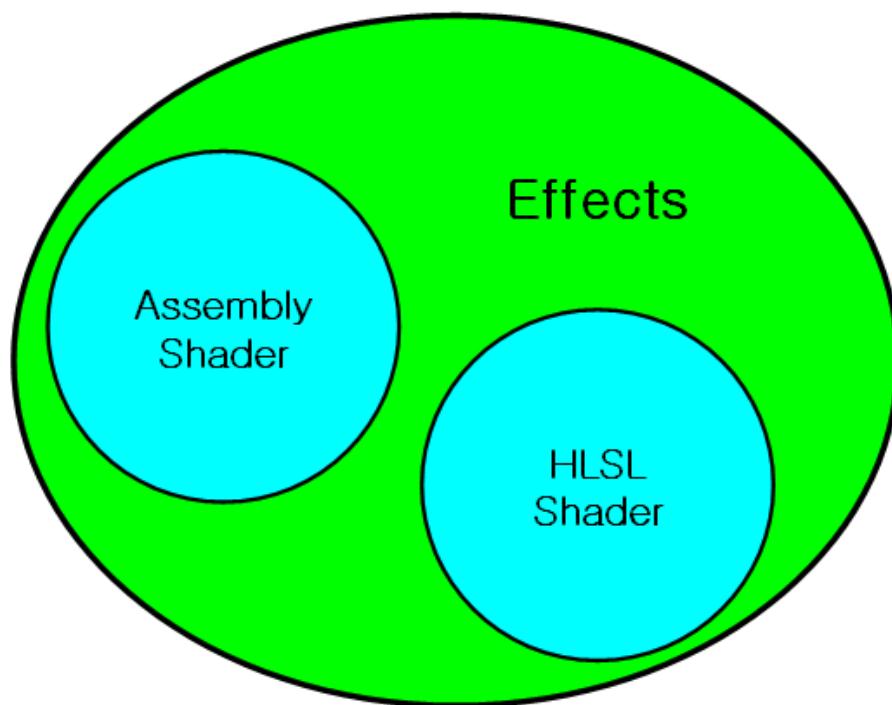
<ILcEffect의 구현과 Glow 효과: [hs6\\_ILcEffect.zip](#)>

## 5 HLSL - ID3DXEffect 응용

### 5.1 HLSL과 ID3DXEffect

우리는 지금까지 정점 쉐이더와 퍽셀 쉐이더를 어셈블리 형식의 저 수준 쉐이더 언어를 C언어와 같은 고 수준 언어 HLSL로 각각 독립적으로 작성해서 적용해왔습니다. 그런데 쉐이더를 사용한다는 것은 프로그램 가능한 파이프라인을 이용하는 것을 의미하고 정점과 퍽셀을 독립적으로 각각 처리하기 보다 동시에 처리하는 것을 염두 해 두는 것입니다. 물론 정점 쉐이더와 퍽셀 쉐이더를 구분해서 컴파일을 하고, 독립적으로 사용하는 것이 각각 다른 그래픽 카드의 성능에 대처할 수 있는 유연성이 있어 보이지만 이것도 통합적으로 작성해서 각 상황에 맞게 처리할 수 있는 방법으로 해결하는 것이 나을 수 있습니다.

현재 대부분의 그래픽 카드들은 퍽셀 쉐이더를 지원하는 경우라면 정점 쉐이더는 당연히 지원이 되고 프로그램에서도 퍽셀 쉐이더를 분리해서 사용할 필요가 없기 때문에 이를 한꺼번에 처리하고 코드의 구현에서 좀 더 편리성을 위한 객체가 필요합니다. 다행히도 마이크로소프트는 Effect 객체를 제공합니다. 이것은 어셈블리 + HLSL + a 이상의 내용을 담고 있어서 고정파이프라인에서 보다 다양한 3D 장면을 연출할 수 있게 해주고 있습니다.



<ID3DXEffect 객체의 역할>

D3D Effect는 위의 그림처럼 저 수준 쉐이더와 고 수준 쉐이더를 기본적으로 지원하고 작업의 편리성을 위한 상수의 공용화, 정점 처리 상태 설정, 퍽셀 처리 상태 설정 등의 옵션(Option)들이

추가된 형태입니다.

기본적인 문법은 이전의 저 수준, 고 수준과 같고 정점 처리와 퍽셀 처리는 각각 작성하고 컴파일 방법을 지정합니다. [ef01\\_basic1\\_hlsl1.zip](#) 예제의 "data/hlsl.fx" 파일은 ID3DXEffect를 위한 쉐이더이며 코드를 보면 정점과 퍽셀 처리의 main 함수 작성은 이전에 배운 HLSL과 동일하고 마지막에 "technique"과 "pass"안에서 컴파일을 지정하고 있음을 볼 수 있습니다.

키워드 "technique" 은 정점과 퍽셀 처리에 대한 단위이고, "pass"는 정점 처리와 퍽셀 처리의 main 함수와 렌더링의 상태 설정입니다.

하나의 테크닉은 여러 패스를 가질 수 있고, 전체 쉐이더 코드는 여러 개의 테크닉을 가질 수 있습니다.

```
technique Tech0
{
    pass P0
    {
        VertexShader = compile vs_1_1 VtxPrc();
        PixelShader = compile ps_1_1 PxlPrc();
    }
    pass P1
    ...
}

technique Tech1
{
...
}
```

이렇게 여러 개의 Pass와 Technique의 지원은 패스에서 여러 개의 쉐이더 함수를 조합해서 하나의 패스를 만들고, 이를 테크닉은 여러 패스를 설정 할 수 있어서 단순한 HLSL보다 조합에 의한 다양한 연출을 실행 프로그램에서 만들지 않고 쉐이더 언어를 작성하는 곳에서 결정을 할 수 있게 됩니다. 또한 반복적인 쉐이더 내용을 분리해 작성할 수 있어서 모듈의 응집력이 높아집니다.

이렇게 작성된 고 수준 쉐이더 언어는 D3DXCreateEffect…() 함수로 컴파일 하며 컴파일과 동시에 ID3DXEffect 객체를 생성합니다.

```
D3DXCreateEffectFromFile( m_pd3dDevice, "data/hlsl.fx"
    , NULL, NULL, dwFlags, NULL
    , &m_pEft, &pErr);
```

HLSL, 쉐이더와 차이점은 컴파일과 동시에 이펙트 객체를 생성하고 있고, 상수 테이블을 만들지 않고 있습니다. 나머지 예러에 대한 처리는 저 수준 쉐이더, HLSL과 동일합니다.

ID3DXEffect 객체는 내부에 상수를 설정할 수 있도록 구성되어 있으며 인터페이스는 상수 테이블에서 사용한 함수들을 그대로 사용할 수 있고 Technique, Pass, 텍스처에 대한 인터페이스가 추가되어 있습니다.

ID3DXEffect로 장면을 연출하는 과정은 SetVector(), SetMatrix(), SetInt() 등의 함수로 상수를 설정하고, SetTechnique() 함수로 Technique을 지정합니다. 다음으로 Begin() 함수를 이용해서 Pass 개수 확인합니다. 마지막으로 Pass()(2003버전), 또는 BeginPass() / EndPass()(2003 이후 버전) 함수 사이에 장면 연출 함수 Draw...()를 호출합니다. Pass(), BeginPass() 함수는 Pass안에 구성된 정점 쉐이더와 픽셀 쉐이더 객체 사용을 지정하는 것과 같습니다.

구체적으로 각 단계에 대한 예를 보이겠습니다. 먼저 행렬, 벡터 등의 상수 값들은 상수 테이블에서 사용했던 방식 그대로 다음과 같이 사용합니다.

```
m_pEft->SetMatrix("m_mtWld", &mtWld);
m_pEft->SetMatrix("m_mtView", &mtView);
m_pEft->SetMatrix("m_mtProj", &mtProj);
```

Pass들의 집합인 Technique을 SetTechnique() 함수로 지정합니다.

```
m_pEft->SetTechnique("Tech0");
```

다음으로 Technique에 지정된 Pass의 숫자를 Begin() 함수로 얻습니다. Pass의 개수가 필요 없다면 NULL 값을 전달합니다.

```
m_pEft->Begin( &nPass, 0 );      // m_pEft->Begin(NULL, 0 );
...
m_pEft->End();
```

Begin() 함수는 End() 함수와 반드시 짹을 이루어야 합니다. 마지막으로 Begin()/End() 함수 사이에 다음과 같이 for 문 또는 직접 인덱스를 사용해서 Pass() 함수 (2003 이후 버전 BeginPass() / EndPass()) 아래에 Draw...() 함수를 호출합니다. Pass(), 또는 BeginPass()/EndPass() 함수는 Technique 안의 Pass에 지정된 정점 쉐이더와 픽셀 쉐이더 사용을 지정하는 것과 같습니다.

```

for(UINT n = 0; n < nPass; ++n)
{
    m_pEft->Pass( n ); // 2003 이후 버전 m_pEft->BeginPass()
    m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLELIST, 1, m_pVtx, sizeof(VtxD));

    // m_pEft->EndPass(); // 2003 이후 버전
}

m_pEft->End(); // m_pEft->Begin() 과 대응

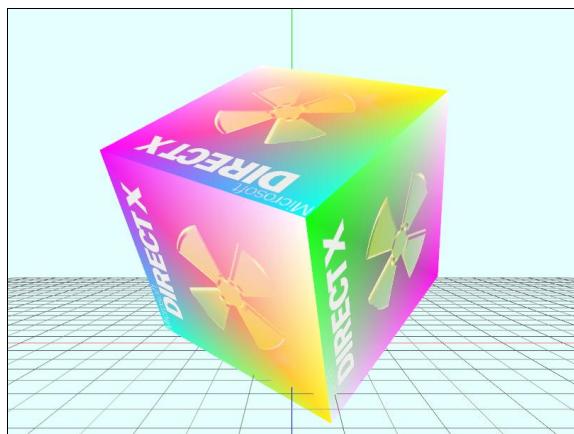
```

End()함수를 호출하면 정점 쉐이더, 퍽셀 쉐이더 사용이 끝나야 되는데 간혹 그래픽 카드에서 처리되지 않을 수 있습니다. 다음과 같이 명시적으로 쉐이더 사용을 해제합니다.

```

m_pDev->SetVertexShader( NULL );
m_pDev->SetPixelShader( NULL );

```



<간단한 ID3DXEffect 사용: [ef01\\_basic1\\_hlsl1.zip](#)>

## 5.2 Multi-Techniques

### 5.2.1 Effect의 상태 설정과 저 수준 쉐이더

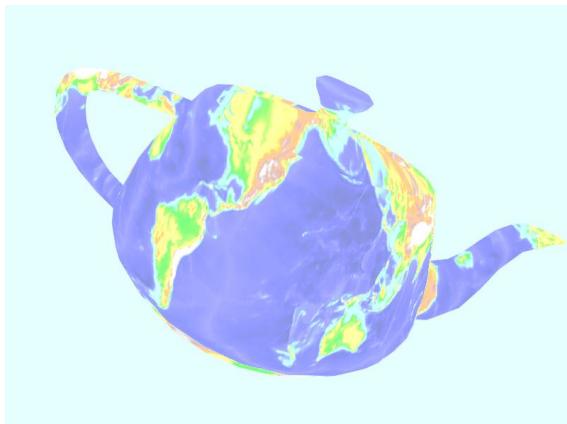
Pass는 렌더링 처리과정의 가장 기본 단위이며 보통 Pass안에 정점 쉐이더 또는 퍽셀 쉐이더 객체를 지정합니다. 그런데 쉐이더 객체를 지정하지 않으면 고정 기능 파이프라인의 정점 처리, 퍽셀 처리, 그리고 기타 렌더링 머신 값을 설정하게 됩니다. 이러한 이점은 다양한 렌더링 환경에 대해서 게임 프로그래머가 특별한 자료 구조를 안 만들어도 되며 범용성이 있어서 다른 게임 프로그램에도 수정 없이 적용할 수 있게 됩니다.

```

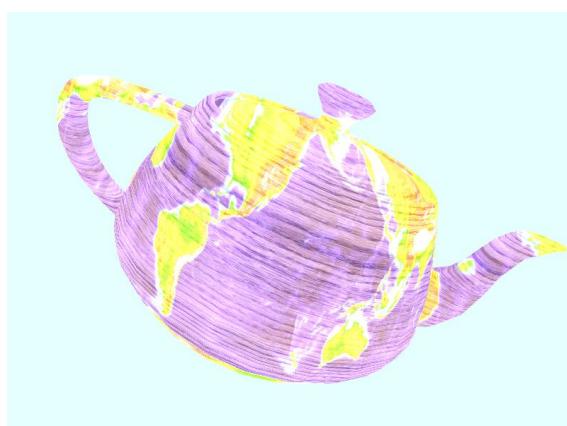
technique Tech0
{
    pass P0
    {
        // Setup Vertex Processing Constant of Rendering Machine
        FOGENABLE      = FALSE;
        LIGHTING       = FALSE;
        CULLMODE       = CCW;

        // Setup Vertex Processing Constant of Rendering Machine
        Sampler[0]     = (SmpDif0);
        ColorOp[0]      = ADDSIGNED;
        ColorArg1[0]    = TEXTURE;
        ColorArg2[0]    = DIFFUSE;
        AlphaOp[0]      = DISABLE;
        ColorOp[1]      = DISABLE;
    }
    pass P1
    ...
}

```



< 고정 기능 파이프라인의 상태 값 설정: [ef01\\_basic1\\_hls12.zip](#) - pass0>

< 고정 기능 파이프라인의 상태 값 설정: [ef01\\_basic1\\_hlsl2.zip](#) - pass 1>< 고정 기능 파이프라인의 상태 값 설정: [ef01\\_basic1\\_hlsl2.zip](#) - pass 2 = 0 + 1>

D3DX의 Effect는 C 언어의 inline assembly 와 같이 저 수준과 고 수준 언어를 혼용해서 사용할 수 있습니다. C 언어는 함수의 중간에 \_\_asm 키워드를 사용해서 inline assembly를 종속적으로 구현하지만 Effect는 정점 쉐이더(Vertex Shader) 객체, 픽셀 쉐이더(Pixel Shader) 객체를 독립적으로 작성하고 이를 Pass안에서 지정하는 형태입니다.

저 수준 쉐이더를 Effect에서 작성하는 방법은 크게 2가지 입니다. 첫 번째 방법은 다음과 같이 technique 밖에서 정점 쉐이더, 또는 픽셀 쉐이더를 작성하고 pass 안에서 지정하는 것입니다.

```
float4x4      m_mtWVP;           // 월드 * 뷰 * 투영 행렬
float3x3      m_mtWld;          // 월드 행렬
float3        m_vcLgt;          // 광원 방향 벡터
```

// 정점 처리의 저 수준 작성

```
VertexShader VtxPrc = asm
{
```

```
    vs_1_1
```

```

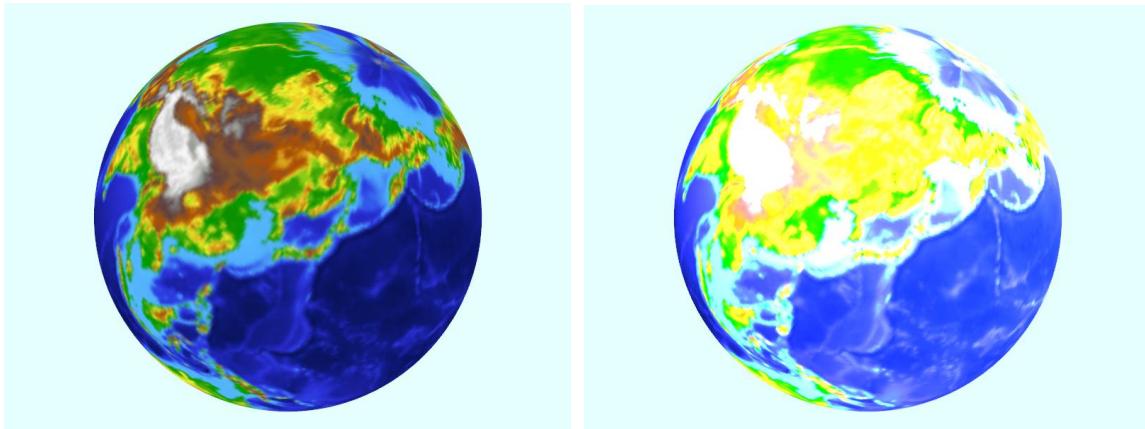
dcl_position    v0
dcl_normal     v1
dcl_texcoord   v2
m4x4 oPos, v0, c0
m3x3 r0,    v1, c4
dp3  r1,    r0,-c8
...
};

// Pixel Processing
PixelShader PxlPrc = asm
{
    ps_2_0
...
};

technique Tech0
{
    pass P0
    {
        // 저 수준 쉐이더에 상수 연결
        VertexShaderConstant4[0] = (m_mtWVP);
        VertexShaderConstant4[4] = (m_mtWld);
        VertexShaderConstant1[8] = (m_vcLgt);

        // 정점 쉐이더 객체 지정
        VertexShader = (VtxPrc);
        // 퍽셀 쉐이더 객체 지정
        PixelShader = (VtxPrc);
    ...
}

```



<Effect 안에서 저 수준 쉐이더1: [ef01\\_basic2\\_asml\\_1.zip](#), [ef01\\_basic2\\_asml\\_2.zip](#)>

저 수준 안에서 사용되는 상수는 pass에서 VertexShaderConstant {1…4} [index], PixelShaderConstant {1…4} [index] 등으로 지정합니다.

예를 들어 4x4행렬을 상수 레지스터 c10에 연결하고자 한다면 VertexShader4[10] = "행렬"; 식으로 작성합니다. 상수 연결과 고정 기능 파이프라인의 연결은 SDK 도움말의 Effect States 부분을 살펴 보기 바랍니다.

저 수준 쉐이더를 사용하는 두 번째 방법은 pass안에 직접 작성하는 것입니다. 상수 설정 등의 Effect의 상태는 이전과 동일합니다.

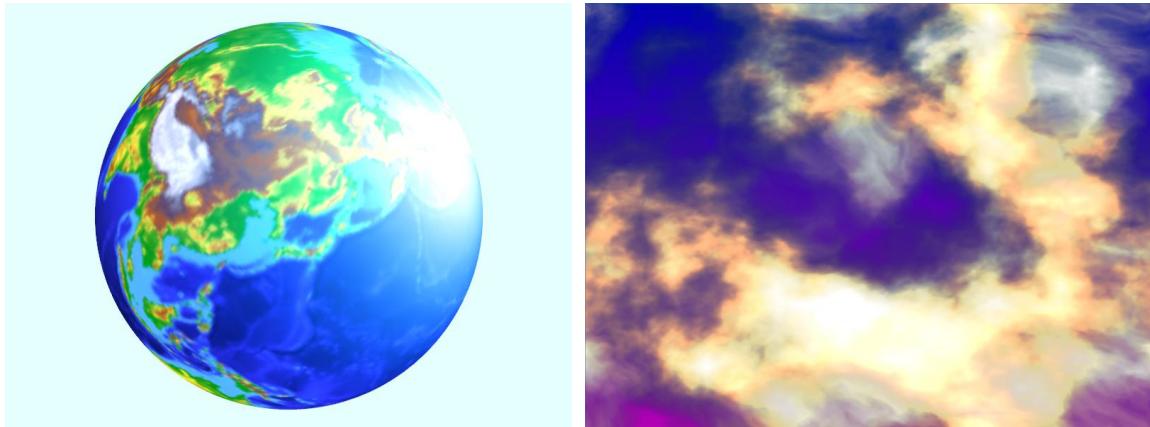
```

float4 m_cv[6];           // 정점 변환 행렬과 텍스처 애니메이션 좌표
float4 m_cp[8];           // 픽셀 쉐이더 상수

technique Tech0
{
    pass P0
    {
        VertexShaderConstant1[0] = (m_cv[0]);
        PixelShaderConstant1[0] = (m_cp[0]);
        ...
        VertexShader = asm
        {
            vs_1_1
        ...
        };
        PixelShader = asm
    }
}

```

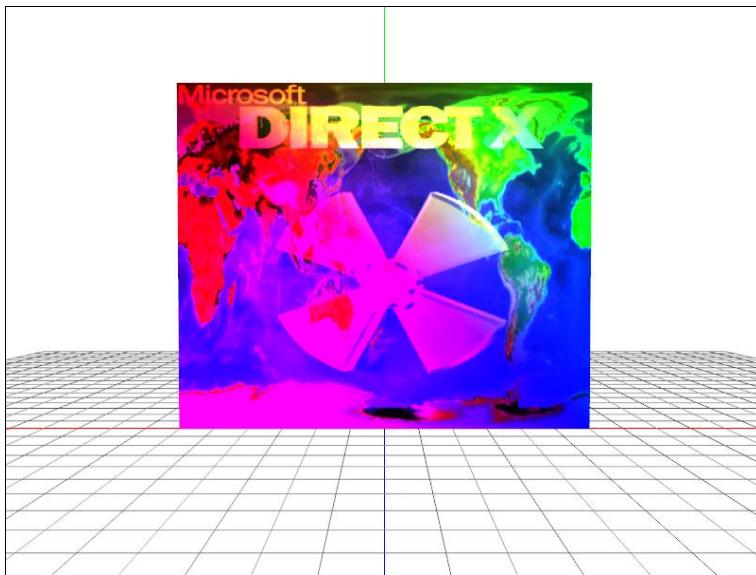
```
{
    ps_1_4
...
}
```



< Effect 안에서 저 수준 쉐이더2: [ef01\\_basic2\\_asm2\\_1.zip](#), [ef01\\_basic2\\_asm2\\_2.zip](#)>

지금까지의 내용을 종합해 보면 technique은 렌더링 상태 머신 값을 설정하고, 특히 쉐이더 객체가 지정되지 않으면 고정 기능 파이프라인의 상태 값을 조정할 수 있습니다. 또한 고 수준으로 작성된 HLSL과 저 수준으로 작성된 쉐이더를 연결해서 사용할 수 있습니다. 또한, 고정 기능, 저 수준, 고 수준 모두를 혼합해서 사용이 가능합니다.

다음은 간단하게 저 수준과 HLSL을 혼합한 예입니다.



<저 수준, HLSL을 혼합해서 사용한 예: [ef01\\_basic3\\_hsl+asm.zip](#)>

### 5.2.2 Multi-pass

앞서 DX의 Effect는 하나의 Technique에 여러 Pass를 가질 수 있다고 했습니다. 이것을 Multi-pass라 합니다. Multi-pass를 사용하면 같은 렌더링 물체에 대해서 서로 다른 렌더링 환경을 가지고 연속해서 그리는 상황에 대해서 여러 편리한 점이 많습니다. 고정 기능 파이프라인에서 Multi-pass와 같은 내용을 구현하려면 코드가 길어지고, 길어진 길이만큼 관리하기가 점점 어려워집니다.

예를 들어 HLSL 기초의 Glow 효과는 렌더링 상태 값을 변경해 가면서 CW, CCW 방식으로 같은 물체를 두 번 렌더링으로 만든 효과인데 HLSL을 사용하더라도 상태 설정 등은 D3D 디바이스의 고정 기능 함수들을 사용했었습니다. 그런데 Effect를 사용하면 다음과 같이 Pass안에서 지정하고 Effect 객체를 사용하는 C/C++ 은 Pass의 숫자를 확인해서 단순히 이들 Pass들을 렌더링 하는 방식이 구성되어 코드의 훨씬 간결하게 됩니다.

```

technique Tech0          // Effect Technique
{
    pass P0           // 모델 렌더링
    {
        CULLMODE      = NONE;
        ...
        VertexShader = compile vs_1_1 VtxPrc0();
        PixelShader  = compile ps_1_1 PxlPrc0();
    }

    pass P1           // Glow 렌더링
    {
        CULLMODE      = CW;
        ...
        VertexShader = compile vs_1_1 VtxPrc1();
        ...
    }
}

// C++
void CShaderEx::Render()
{
    ...
    // 상수 연결
    hr = m_pEft->SetMatrix("m_mtWld", &m_mtWld);
    ...
    // Technique 지정
    hr = m_pEft->SetTechnique("Tech0");
}

```

```

// Pass 개수 확인
UINT nPass=0;
hr = m_pEft->Begin( &nPass, 0 );

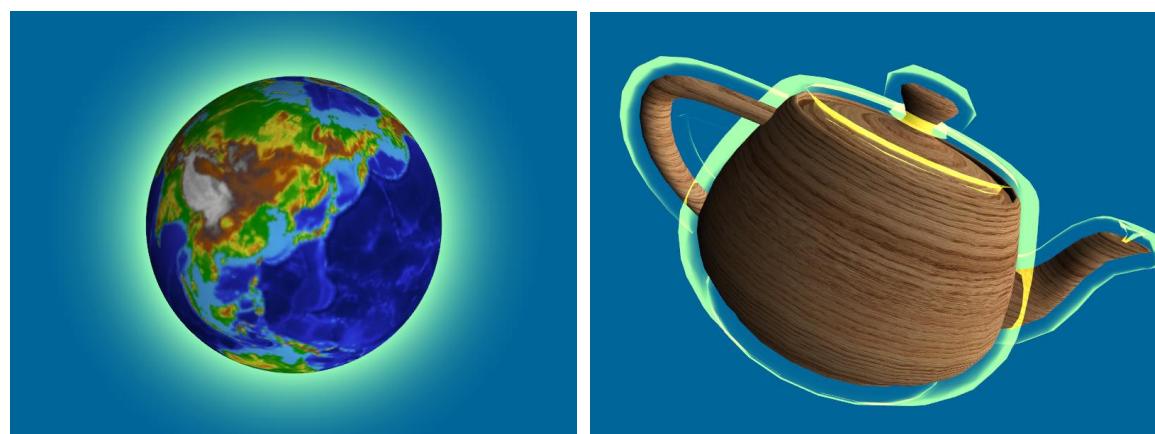
// Pass 수 만큼 같은 물체를 렌더링
for(UINT n = 0; n < nPass; ++n)
{
#ifndef D3DX_SDK_VERSION >21
    hr = m_pEft->BeginPass( n );
#else
    hr = m_pEft->Pass( n );
#endif

hr = m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLESTRIP, ...);

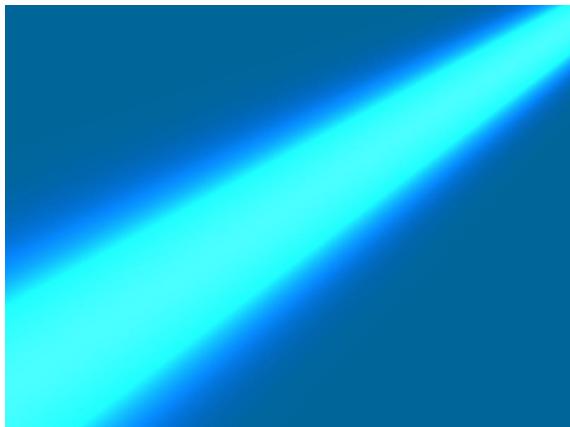
#ifndef D3DX_SDK_VERSION >21
    hr = m_pEft->EndPass();
#endif
}

hr = m_pEft->End();

```



<법선 벡터를 이용한 효과: [ef02\\_pass1\\_glow1.zip](#), [ef02\\_pass1\\_glow2.zip](#)



<범선 벡터를 이용한 효과: [ef02\\_pass1\\_laser.zip](#)>

이처럼 Multi-Pass의 사용은 코드를 간결하게 만들고 제어를 Effect에서 작성이 되어 게임의 기본 골격은 프로그래머가 만들고 효과에 대한 처리와 상태 값들은 쉐이더 언어를 알고 있는 기획자가 만들게 되어 전체적인 게임의 연출의 완성도가 높아지고 작업 또한 적절히 분배 되는 장점이 있습니다.

현재의 그래픽 카드들은 한번의 렌더링에 대한 속도가 모니터의 Hz보다 높게 나오는 것이 많습니다. 사람의 눈은 24~30 프레임 정도이기 때문에 고성능 그래픽 카드에서 같은 물체를 다른 환경에서 반복적으로 렌더링 해서 시각적 효과를 높이는 방법이 많이 사용되고 있습니다. 부드러운 그림자, Post Effect 등은 대표적으로 같은 물체를 반복해서 만드는 예이며 이들은 다음 과정에서 다시 살펴 보겠습니다.

간단하게 Multi-Pass를 사용해서 Post Effect의 하나인 Blur 효과를 구현해 보겠습니다. Blur 효과는 이후 과정에도 나오지만 인접한 픽셀에 가중치를 주고 이 가중치를 인접한 픽셀에 곱한 다음 전부 더해서 최종 색상을 정하는 방법입니다.

$$\text{Blur 효과 최종 색상} = \frac{\sum W_i * \text{Pixel}_i}{\sum W_i}$$

전체 장면을 저장한 텍스처를 사용하기 때문에 정점의 형식은 RHW를 사용하거나 뷰 행렬과 투영 행렬을 사용하게 되어 정점 처리에 대한 쉐이더는 입력 값을 그대로 출력으로 사용하게 되어 다음과 같이 간단한 쉐이더로 작성합니다.

```
VertexShader VtxPrc = asm
{
    vs_1_1
```

```

dcl_position v0
dcl_color0  v1
dcl_texcoord v2
mov oPos, v0
mov oD0, v1
mov oT0, v2
};


```

생동감 있는 Blur 효과를 만들기 위해서 먼저 전체 장면을 저장한 텍스처를 그대로 화면에 그리는 데 절반(1/2)으로 낮추어서 그립니다.

```

float4 Px1Prc0(float4 Dif : COLORO, float2 Tx0 : TEXCOORD0) : COLOR
{
    float4 Out= tex2D(sampTex, Tx0);
    Out *= 0.5f;
    return Out;
}

```

다음으로 장면을 저장한 텍스처에 Blur 효과를 적용합니다. 가중치(weight: w)는 Gaussian 분포 함수( $I * \exp(-x*x/\delta)$ )를 사용하게 되어 쉐이더의  $\exp()$  함수를 이용합니다.

```

// Blur 효과
float4 Px1Prc1(float4 Dif : COLORO, float2 Tx0 : TEXCOORD0) : COLOR
{
    int i=0; int iMax=4; float4 Out= 0.0f;

    // 인접한 픽셀에 가중치를 주고 이들을 더함
    // 먼저 x 방향으로 Blur 효과 적용
    for(i=-iMax; i<=iMax; ++i)
    {
        float2 Tx = Tx0;
        Tx.x += (i * m_fDeviation)/1024.f;
        float4 d = tex2D(sampTex, Tx);
        // Gaussian 분포 함수  $f(x) = I * \exp(-x*x/\delta)$ 
        float e = i*i;
        e = -e/16.0f;
        Out += d * 1.0f* exp( e );
    }
}


```

```

}

// y 방향으로 Blur 효과 적용
for(i=-iMax; i<=iMax; ++i)
{
    float2 Tx = Tx0;
    Tx.y += (i * m_fDeviation)/1024.0f;
    float4 d = tex2D(sampTex, Tx);
    float e = i*i;
    e = -e/16.0f;
    Out += d * 1.0f* exp( e );
}
...

```

Technique에서 Blur 효과를 적용하는 Pass는 Alpha Blending을 활성화하고 Source와 Dest는 전부 ONE으로 설정합니다.

```

technique Tech
{
    pass P1
    {
        AlphablendEnable= TRUE;
        SRCBLEND      = ONE;
        DESTBLEND     = ONE;
        ...
        VertexShader = (VtxPrc);
        PixelShader   = compile ps_2_0 PxlPrc1();
    }
}

```



<인접 픽셀을 이용한 Blur 효과: [ef02\\_pass2\\_blur.zip](#)>

[ef02\\_pass2\\_blur.zip](#)를 실행하면 시간에 대해서 Blur 효과가 동적으로 적용되는 것을 볼 수 있습니다.

### 5.3 2D Sprite

HLSL이나 Effect는 D3D에서도 고급 내용이기 때문에 처음에는 익숙하지 않을 수 있습니다. 3D 프로그램은 게임 제작 등을 통해서 개인의 실력이 발전하는데 프로젝트가 준비 되지 않고 공부하는 과정이라면 Effect를 연습하기 위해 2D 게임에 대한 Sprite를 Effect로 만드는 것을 추천합니다.

DX 10이상은 쉐이더가 기본입니다. Embedded 환경에서 대부분 사용되는 OpenGL ES는 버전 2.0부터 GLSL을 사용하고 GLSL은 HLSL과 많이 비슷합니다. 따라서 2D 게임을 위한 Sprite를 Effect로 작성하는 것은 꼭 필요한 일이라 할 수 있습니다.

ID3DXSprite는 크기, 회전, 이동, 색상을 설정할 수 있는 Sprite입니다. 여기에 단색화를 추가한 Sprite를 쉐이더와 Effect로 구성해 봅시다.

먼저 입력 값을 다음과 같이 단색화, 색상 적용 값, 텍스처 샘플러를 지정합니다.

```
int      m_bMono;           // 단색화 사용
```

```
float4 m_Diff;           // 색상 값
sampler smp0 : register(s0); // 샘플러
```

만약 Sprite의 정점이 RHW로 구성되어 있다면 정점 처리 과정은 특별히 처리할 일이 없으므로 입력 값을 그대로 출력하도록 작성합니다.

```
VertexShader VtxProc = asm      // 정점 처리: 입력 값을 그대로 출력
{
    vs_1_1
    dcl_position v0
    dcl_texcoord v1
    mov oPos, v0
    mov oT0 , v1
};
```

픽셀 처리 과정의 기본은 최종 색상을 주어진 색상과 텍스처의 색상의 곱으로 결정하는 것입니다. 그런데 단색화의 요구가 있을 수 있으므로 이 경우에는 rgb는 외부에서 주어진 rgb으로 사용하고 알파는 텍스처의 알파 값을 사용합니다.

```
float4 Px1Proc(float4 Pos0: POSITION
                , float2 Tex0: TEXCOORD0) : COLOR0
{
    float4 Out= tex2D(smp0, Tex0);

    Out *= m_Diff;
    if(0 != m_bMono)           // 단색
    {
        Out.a *= m_Diff.a;    // 알파는 텍스처의 알파 사용
        Out.r= m_Diff.r;      // 입력 값 색상 사용
        Out.g= m_Diff.g;
        Out.b= m_Diff.b;
    }

    return Out;
}
```

Technique에서 렌더링 상태 머신 같은 알파 블렌딩을 활성화 하고 Source와 Dest의 알파 값은

각각 SRCALPHA, INVSRCALPHA를 사용합니다. 또한 U, V 값이 [0,1] 범위에서만 유효하도록 Address Mode를 Clamp로 설정합니다.

```
technique Tech
{
    pass P0
    {
        CULLMODE           = NONE;
        ALPHABLENDENABLE = TRUE;
        SRCBLEND          = SRCALPHA;
        DESTBLEND          = INVSRCALPHA;
        ADDRESSU[0]         = CLAMP;
        ADDRESSV[0]         = CLAMP;
        VertexShader = (VtxProc);
        PixelShader   = compile ps_1_1 Px1Proc();
    }
};
```

게임에서 단색화가 필요한 경우는 주로 그림자를 그릴 때입니다. 또한 2D 게임은 확대/축소에서도 2D 특유의 느낌을 살리기 위해 픽셀의 필터링을 적용하지 않지 않는 경우가 많이 있습니다. 그림자를 그릴 때만 부드럽게 적용하도록 필터링을 설정합니다.

```
if(bMono)          // 단색일 경우 필터링 적용
{
    m_pDev->SetSamplerState(0,D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
    m_pDev->SetSamplerState(0,D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
    m_pDev->SetSamplerState(0,D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
}
else
{
    m_pDev->SetSamplerState(0,D3DSAMP_MAGFILTER, D3DTEXF_NONE);
    m_pDev->SetSamplerState(0,D3DSAMP_MINFILTER, D3DTEXF_NONE);
    m_pDev->SetSamplerState(0,D3DSAMP_MIPFILTER, D3DTEXF_NONE);
}
```



<쉐이더, Effect로 만든 2D Sprite: [ef03\\_sprite1\\_soft.zip](#)>

직소(Jigsaw) 게임은 특정한 패턴으로 원본 이미지를 잘라내어야 합니다. 이것을 그래픽에서 작업한다면 광장한 노동력이 필요합니다. 그런데 알파 맵을 만든다면 수작업으로 이미지를 잘라내는 일들을 줄일 수가 있습니다. 이미지의 특정한 영역을 잘라내도록 알파 값을 가진 텍스처를 Alpha map(알파 맵)으로 부르도록 합시다. 알파 맵을 Sprite에 적용하기 위해서 다음과 같이 샘플러를 하나 더 추가합니다.

```
int      m_bTx1;           // 알파 맵 사용
int      m_bMono;          // 단색화 사용
float4   m_Diff;           // 외부 색상 값
sampler smp0 : register(s0); // 원본 이미지용 샘플러
sampler smp1 : register(s1); // 알파 맵 용 샘플러
```

알파 맵 또한 UV를 갖도록 정점 구조체를 수정합니다.

```
struct VtxRHWUV1
{
    VEC2    p;        FLOAT    z;        FLOAT    w;
    VEC2    t0;       // 원본 이미지 좌표
    VEC2    t1;       // 알파 맵 이미지 좌표
    ...
    enum {FVF = (D3DFVF_XYZRHW|D3DFVF_TEX2), };
};
```

정점 처리 과정은 이전의 Sprite처럼 입력 값을 그대로 출력 값으로 복사하도록 작성합니다.

```
VertexShader VtxProc = asm      // 정점 처리: 입력 값을 그대로 출력
...
dcl_texcoord0 v1      // 원본 좌표
dcl_texcoord1 v2      // 알파 맵 좌표
mov oT0 , v1
mov oT1 , v2
...
```

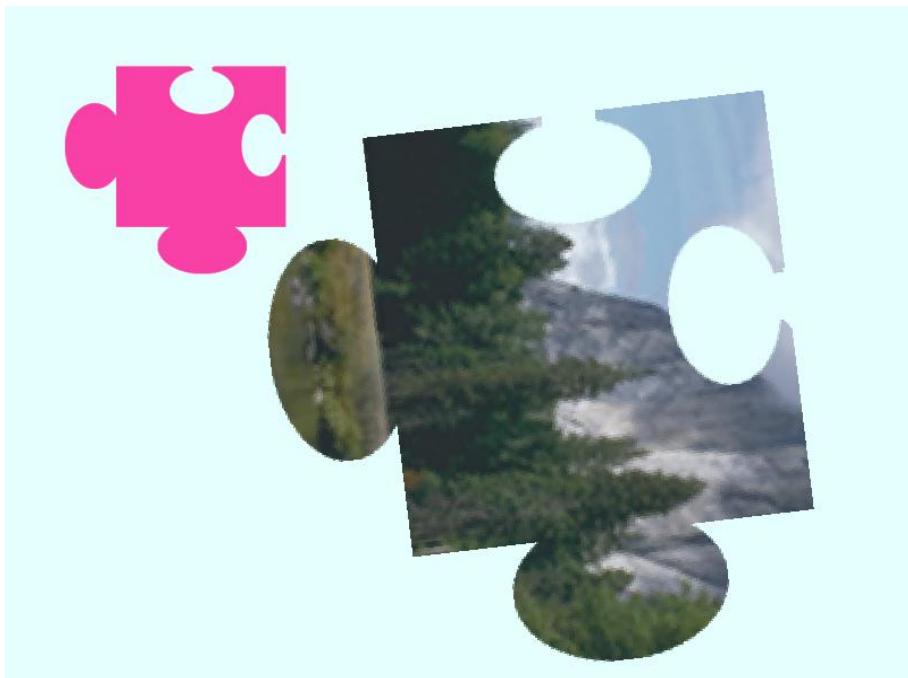
픽셀 처리과정은 알파 맵 적용, 단색화 두 개의 선택이 있습니다. 먼저 알파 맵 적용의 요구가 있으면 원본 텍스처와 알파 맵을 곱셈(Modulate)로 처리합니다.

```
float4 PxlProc( float4 Pos0: POSITION
                 , float2 Tex0: TEXCOORD0      // 원본 입력 좌표
                 , float2 Tex1: TEXCOORD1      // 알파 맵 입력 좌표
                ):COLOR0
{
    float4 Out= 0;
    float4 t0 = tex2D(smp0, Tex0);
    float4 t1 = tex2D(smp1, Tex1);

    if(0 != m_bTx1)           // 알파 맵 사용이 있으면 두 텍스처를 곱함
        Out = t0 * t1;
    else
        Out = t0;

    Out *= m_Diff;
    if(0 != m_bMono)          // 단색
    {
        Out.a *= m_Diff.a;    // 알파는 텍스처의 알파 사용
        Out.r= m_Diff.r;      // 입력 값 색상 사용
    }
    ...
}
```

ef03\_sprite2\_hard를 실행하면 다음과 같이 알파 맵이 적용된 단색화 부분과 텍스처 이미지가 출력되는 것을 볼 수 있습니다.



<알파 맵이 적용된 Effect로 만든 2D Sprite: [ef03\\_sprite2\\_hard.zip](#)>

이처럼 2D Sprite를 Effect 만들면 사용의 요구에 따라서 능동적으로 대처할 수 있으므로 본격적으로 게임 제작에 돌입하기 전에 Sprite를 Effect로 만들어 보도록 하기 바랍니다.

## 5.4 Effect와 조명

쉐이더와 HLSL의 기본적인 내용을 알고 있다면 게임 프로그램의 3D 장면 연출에서 가장 중요하고 또한 가장 먼저 쉽게 적용할 수 있는 부분이 조명입니다. 조명은 같은 모델이라도 전혀 다른 느낌을 연출할 수 있음을 고정 기능 파이프라인과 저 수준 쉐이더, 그리고 HLSL에서 충분히 경험했습니다. D3D의 조명을 간단히 정리하면 분산 조명은 램버트 확산을 배경으로 만들어져 있고, Specular 조명은 풍 반사에 기반을 두고 있습니다. 또한 이 둘은 고정 기능 파이프 라인에서 분산 조명은 구로 쉐이딩(Henri Gouraud)으로, 그리고 스페큘러 쉐이딩으로 불리어지고 있습니다.

3D 기초 시간에 고정 기능 파이프라인은 정점을 기준으로 모든 것을 처리하고 있어서 정점의 숫자가 적은 경우 원하는 조명의 효과를 보지 못한다고 했습니다. 이것을 해결하기 위해서 퍽셀 처리 과정에서 조명에 대한 계산이 필요하다고 했습니다.

이 강의에서는 조명에 대한 모든 것을 완성한다는 의미에서 퍽셀 쉐이더 기반의 조명 처리뿐만 아니라 조명에 관련된 Toon 쉐이딩을 다시 살펴보고, 마지막으로 NPR 중에서 tonal art의 하나인 Hatching을 살펴 보겠습니다.

### 5.4.1 분산 조명

이전 장에서 공부 했듯이 분산 조명은 램버트 확산에 기반을 둔 D3D의 분산 조명은 램버트 확산에 기초를 두고 있습니다. 램버트 확산은 반사되는 빛의 세기(Intensity)를 빛의 방향과 정점의 법선 벡터와의 내적을 통해서 구합니다. 이를 구하는 공식은 다음과 같습니다.

반사 밝기 =  $\text{dot}(\text{정점의 법선 벡터 } N, \text{ 빛의 방향 벡터 } L)$

이 공식을 그대로 적용하게 되면 최종 색상의 범위는 -1.0 ~ +1.0 가 되므로 적정한 명도를 만들기 위해 우리는 다음과 같은 공식을 사용했습니다.

반사 밝기 =  $a * \text{dot}(N, L) + b$  또는  $(a + \text{dot}(N, L)) * b$

이 공식을 다음과 같은 HLSL로 쉽게 바꾸는 것도 연습했었습니다.

```
float3x3      m_mtRot;           // 회전 행렬
float3        m_vcLgt;           // 빛의 방향 벡터
...
float3 N = mul(Nor, m_mtRot);   // 법선 벡터의 회전 변환
float3 L = -m_vcLgt;           // 빛의 방향 벡터 반전
float4 D = (0.5f + dot(N, L)) * 0.6f; // Lambert 공식으로 밝기 설정
```



<분산 조명 밝기: [ht11\\_lam1\\_basic.zip](#)>

분산 조명의 밝기를 정점 처리에서 결정했는데 이것을 픽셀 처리 과정에서 계산해 보도록 합시다. 픽셀 처리과정에서 조명 효과를 계산하는 이유는 정점 처리과정은 조명 효과를 먼저 계산해서 정

점의 색상에 반영을 하고 이것을 다시 래스터 과정에서 정점 사이의 색상을 구로 쉐이딩으로 보간하기 때문에 좀 더 자연스러운 조명 효과를 만들기 위해서 퍽셀 처리과정에서 조명 효과를 처리하도록 하는 것입니다.

정점 처리 과정에서 정점의 법선 벡터를 회전 변환만 적용하고 변환한 법선 벡터를 퍽셀 쉐이더로 전달합니다. 퍽셀 쉐이더로 전달할 때 지정된 법선 벡터의 레지스터가 없으므로 텍스처 좌표 레지스터를 사용합니다. 이로 인해서 전달된 법선 벡터는 소속이 텍스처 좌표여서 GPU는 인접한 법선들과 다음과 같은 공식을 가지고 내부에서 선형 보간을 수행합니다.

$$\hat{n} = (1-w) * \hat{n}_i \bullet \hat{L} + w * \hat{n}_f \bullet \hat{L}$$

(w 는 보간에 필요한 비중(Weight))

텍스처 레지스터를 사용할 때 0번부터는 원래의 텍스처 좌표로 주로 사용되기 때문에 인덱스 끝 번호 7부터 현재 이용되지 않는 텍스처 좌표 레지스터를 사용할 수 있도록 다음과 같이 출력 구조체를 작성 합니다.

```
struct SVsOut
{
    float4 Pos : POSITION; // 출력 위치(oPos)
    float2 Tx0 : TEXCOORD0; // 출력 텍스처 좌표(oT0)
    float3 Nor : TEXCOORD7; // 텍스처 좌표를 변환된 법선의 좌표로 사용
};
```

조명에 대한 정점 처리는 법선 벡터의 회전만 변환합니다.

```
SVsOut VtxPrc( float4 Pos : POSITION, // 입력 위치
                 float4 Nor : NORMAL, // 입력 법선 벡터
                 float2 Tx0 : TEXCOORD0 // 입력 텍스처 좌표
...
                 float3 N = mul(Nor, m_mtRot); // 정점의 법선 벡터의 회전 변환
                 Out.Pos = P; // 출력 위치 복사
                 Out.Nor = N; // 출력 법선 벡터
                 Out.Tx0 = Tx0; // 출력 텍스처 좌표
                 return Out;
```

정점에서 처리했던 분산 조명을 퍽셀 처리과정에서 연산합니다. 최종 색상은 물체의 텍스처와 혼합해서 출력하되 알파 값은 텍스처의 알파 값을 사용하도록 합니다.

```

float4 Px1Prc(SVsOut In) : COLOR
{
    float4 Out = 0;                                // 출력 색상
    float3 N = normalize(In.Nor);                  // 법선 벡터의 정규화
    float3 L = -m_vcLgt;                           // 빛의 방향 벡터의 반전

    Out = (0.5f + dot(N, L)) * 0.6;               // Lambert
    Out.a = 1.0f;                                  // 텍스처의 알파를 사용하도록 1로 설정
    Out *= tex2D( SampDiff, In.Tx0 );             // 텍스처의 색상과 곱셈
    return Out;
}

```



<분산 조명과 텍스처: [ht11\\_lam2\\_diffuse.zip](#)>

만약 여러분이 Effect를 사용하지 않고 분산 조명을 처리한다면 정점 쉐이더, 퍽셀 쉐이더 객체를 생성하고 상수 값들을 설정하기 위해 상수 테이블을 사용해야 하겠지만 Effect의 사용으로 인해서 이들 상수 값들이 어느 처리에서 적용되는지 큰 고민 없이 편하게 작성할 수 있게 되었습니다

분산 조명에 대한 강의를 맷는 의미로 다중 조명 처리를 만들어 보도록 하겠습니다. 반사의 밝기와 빛의 색상을 적용하면 반사된 빛의 최종 색상은 다음과 같이 설정할 수 있습니다.

$$\text{분산 조명 색상} = \sum \text{dot}(N, L_i) * \text{Lighting Color}_i$$

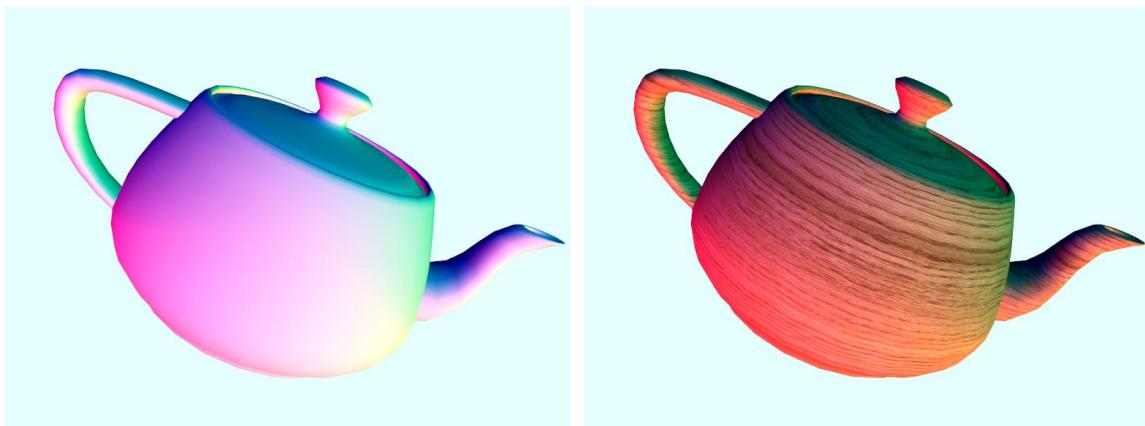
이것을 쉐이더로 구현하면 for문을 통해서 분산 조명과 빛의 색상을 반복적으로 적용해서 최종 색상을 만들어냅니다.

```
float3 m_LgtDir[4]; // 광원 방향 벡터 배열
```

```

float4 m_LgtDif[4]; // 광원 색상 배열
...
float4 Out = 0; // 출력 색상
...
for(int i=0; i<4; ++i)
{
    float4 t=0;
    t = 0.25f + dot(N, -m_LgtDir[i]) * .8f; // Lambert
    t *= m_LgtDif[i];
    Out +=t;
}

```



<여러 광원을 이용한 분산 조명 효과: [ht11\\_lam3\\_multi.zip](#) - 'T', 'R' 키>

여러 강원을 이용한 분산 조명 효과 예제나 그 이전의 분산 조명과 텍스처 예제는 퍽셀 처리 과정에서 밝기를 계산했습니다. 그런데 이들은 정점 처리 과정에서 계산한 것과 뚜렷한 차이를 느끼지 못할 수 있습니다. 원래 정점 사이의 분산 조명의 밝기를 정확하게 계산하려면 사원수의 보간에서 유도된 공식을 이용해서 다음과 같이 법선 벡터를 구해야 합니다.

$$\hat{n} = \frac{1}{\sin \theta} [\sin(\theta(1-w)) \hat{n}_i + \sin(\theta w) \hat{n}_f], \quad (\theta \text{는 } \hat{n}_i \text{과 } \hat{n}_f \text{의 사이 각, } w=[0,1] \text{의 비중 값})$$

그런데 두 법선 벡터의 사이 각이 작다면  $\sin \theta \approx \theta$ ,  $\sin \theta t \approx \theta t$ ,  $\sin \theta(1-t) \approx \theta(1-t)$ 이 되어 법선 벡터는 좀 더 간단한 형태가 됩니다.

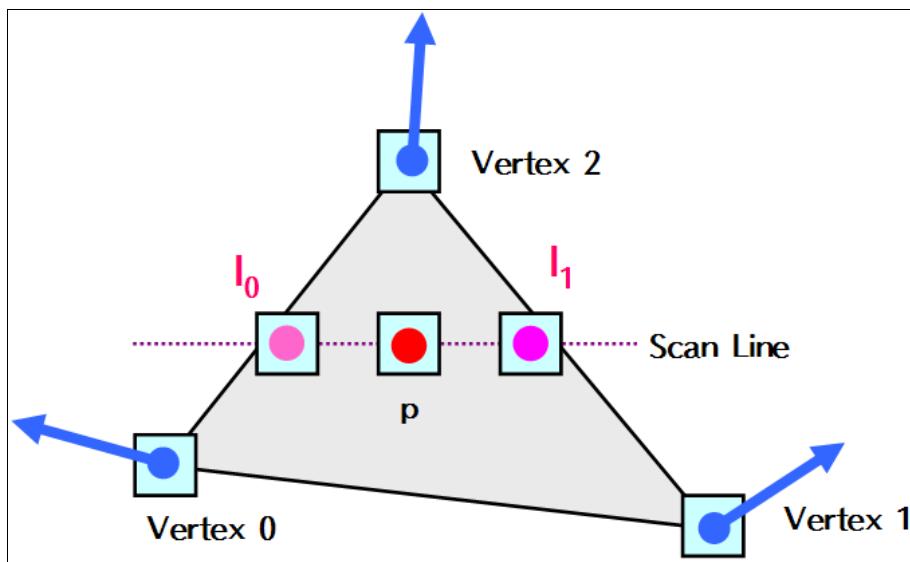
$$\hat{n} = (1-w) * \hat{n}_i + w * \hat{n}_f$$

이것은 법선 벡터를 텍스처 좌표로 넘길 때 GPU가 계산한 법선 벡터와 같습니다. 이로 인해서 분

산 조명을 정점 처리 과정에서 계산한 것과 픽셀 처리 과정에서 계산 결과가 크게 차이가 나지 않는 것입니다.

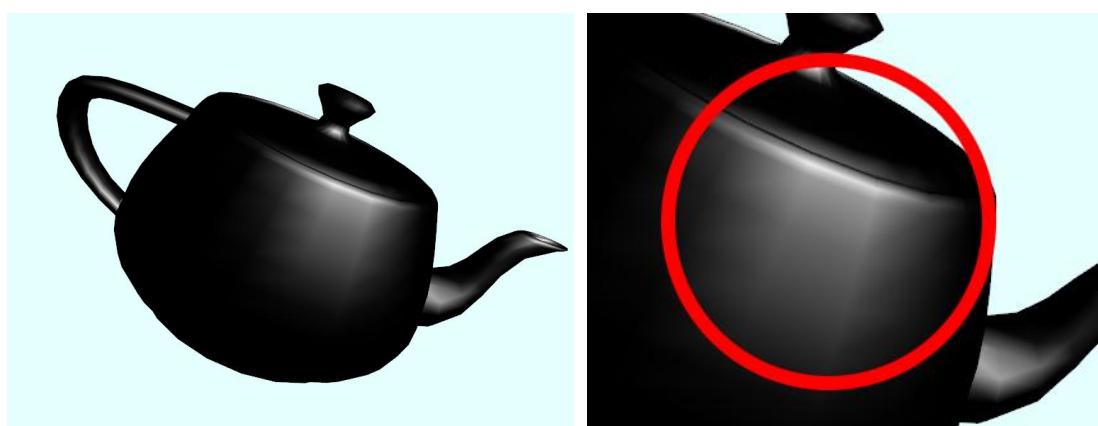
### 5.4.2 Specular 효과

구로 쉐이딩을 다시 살펴 보면 그림과 같이 3개의 정점으로 구성된 삼각형에서  $p$  위치의 색상은 정점 0과 정점 2로 보간된  $I_0$  색상과 정점 1과 정점 2로 보간된  $I_1$  색상을 다시 보간 해서 구해 집니다.



<정점 처리 과정의 구로 쉐이딩: 색상을 먼저 계산하고 이들을 보간>

이것은 분산 조명 효과에서는 거의 문제가 되지 않지만 Specular 조명 효과에서는 정점의 개수가 적으면 다음 그림과 같은 상태를 볼 수 있습니다.

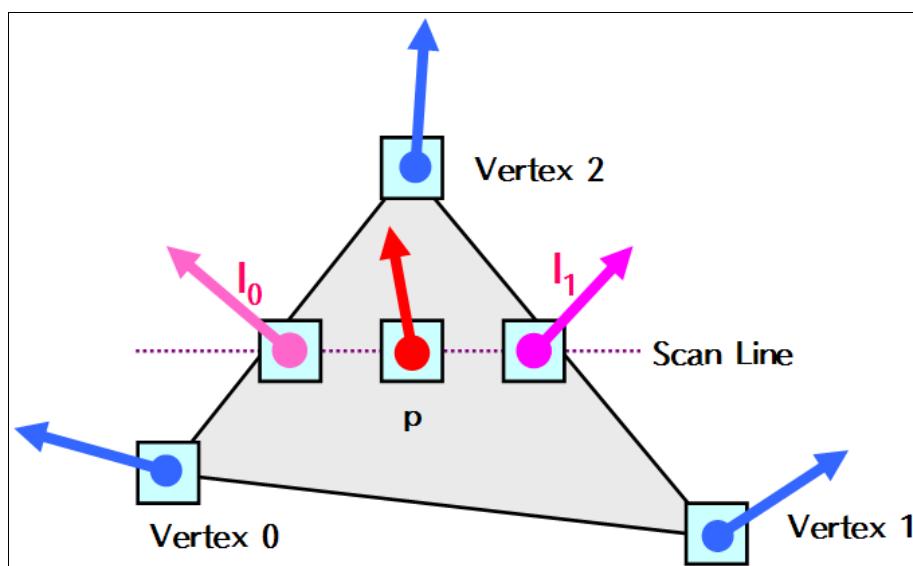


<정점 쉐이더를 사용한 Specular 효과: [ht12\\_spec1\\_vtx.zip](#)>

이것은 정점의 Specular 효과를 점 p에 해당하는 법선 벡터와 시선 벡터로 구하지 않고 정점에서만 계산하고 p 점에서는 이 값을 선형 보간 하기 때문입니다.

그리고 정점 처리과정에서 계산되는 Specular 값은 Diffuse에 반영되고 Diffuse 색상 값은 래스터 처리 후에는 [0, 1]으로 정규화되어 있어서 1보다 큰 빛의 반사 값을 만들어 놓아도 픽셀 처리 과정에서 이 값을 사용할 수 없게 되어 강렬한 빛의 효과를 만들기가 어렵습니다.

픽셀 p의 위치에서 정확한 Specular 효과를 만들기 위해서는  $I_0$ 에서 법선 벡터를 정점 0과 정점 1의 법선의 보간을 만들고  $I_1$ 에서는 법선 벡터를 정점 1과 정점 2의 법선의 보간으로 만듭니다. 그리고  $I_0$  와  $I_1$ 의 법선들을 가지고 픽셀 p의 위치에 맞는 법선 벡터를 구해서 이것을 Specular 공식에 적용해야 합니다.



<정점 법선 벡터를 텍스처 좌표로 픽셀 처리로 넘기기>

현재로서는 이렇게 정확하게 계산할 수 있는 GPU는 그리 많지 않습니다. 대신 픽셀 쉐이더 2.0 이상을 사용하면 어느 정도 이를 구현할 수 있는데 방법은 정점의 법선 벡터를 텍스처 좌표로 저장해서 픽셀 처리기로 넘기는 것입니다. 이렇게 하면 픽셀 처리기는 입력 받은 값은 법선 벡터 값이지만 텍스처 좌표이기 때문에 각각의 픽셀에 대해서 이 값을 선형 보간을 합니다.

선형 보간된 이 텍스처 좌표 값은 크기가 1이 아니므로 법선 벡터로 사용하기 위해서 꼭 정규화를 해야 합니다. 또한 시선 벡터도 텍스처 좌표로 픽셀 처리로 넘기고 픽셀 처리 과정에서 정규화해서 사용합니다.

지금까지의 내용을 구현해 보도록 하기 위해서 먼저 정점 처리과정에서 Specular 효과에 대한 Phong 반사를 작성해 봅시다. 이전에 저 수준, HLSL로 해봤기 때문에 지금은 어려움이 없습니다.

```
float3x3      m_mtRot;           // 법선 벡터
float3        m_vcLgt;           // 빛의 방향 벡터
```

```

float3 m_vcCam;           // 카메라 위치
float    m_fShrp;          // Sharpness

// 정점 처리 프로세스
SVsOut VtxPrc( float3 Pos : POSITION, // 입력 위치
                float3 Nor : NORMAL   // 입력 법선 벡터
...
// 시선 벡터 = 정규화(카메라 위치 - 정점 위치)
float3 E = normalize(m_vcCam - P);

float3 N = normalize(mul(Nor, m_mtRot));           // 법선 벡터의 회전
float3 L = normalize(-m_vcLgt);                     // 빛의 방향 반전
float3 R = normalize( 2 * dot(N, L) * N - L);     // 반사 벡터
float4 S = pow( max(0, dot(R, E)), m_fShrp);     // Phong 반사 세기

```

<[ht12\\_spec1\\_vtx.zip](#)>

다음으로 정점 처리에서 구현한 Specular를 퍽셀 처리에서 구현하기 위해서 법선 벡터와 시선 벡터를 퍽셀 처리기가 보간할 수 있도록 다음과 같이 정점 처리의 출력에 대한 구조체에 텍스처 좌표 Semantic을 사용해서 정점의 법선 벡터와 시선 벡터를 저장할 수 있도록 구성합니다.

```

struct SVsOut
{
    float4 Pos : POSITION; // 출력 위치
    float3 Nor : TEXCOORD6; // 변환된 법선 벡터를 텍스처 좌표 6에 저장
    float3 Eye : TEXCOORD7; // 시선 벡터를 텍스처 좌표 7에 저장
};

```

정점 처리 과정에서는 단순하게 법선 벡터를 회전 변환 하고, 시선 벡터를 구해서 출력 구조체 변수에 저장합니다.

```

SVsOut VtxPrc(...)
...
// 시선 벡터 = 정규화(카메라 위치 - 정점 위치)
float3 E = normalize(m_vcCam - P);
float3 N = mul(Nor, m_mtRot); // 법선 벡터의 회전
...

```

```

Out.Eye = E;      // 시선 벡터 복사
Out.Nor = N;      // 변환된 법선 벡터 복사
return Out;

```

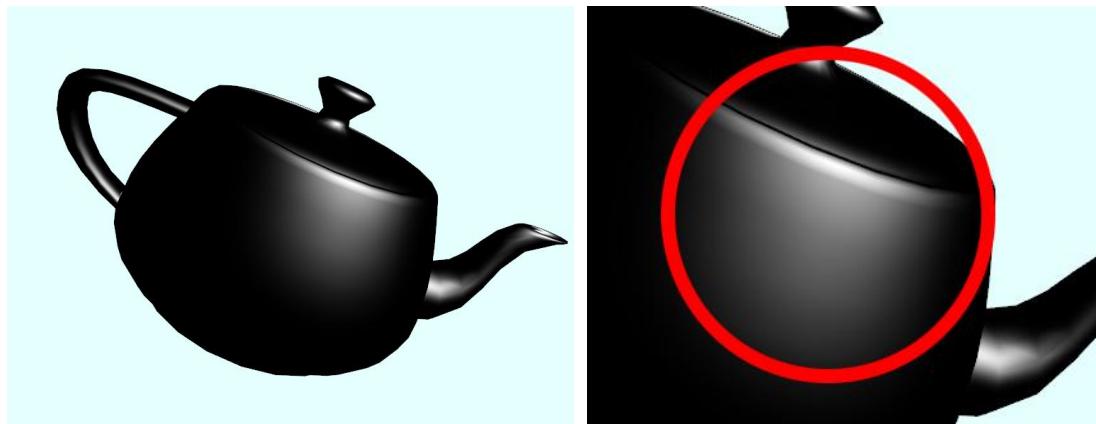
픽셀 처리 과정에서 먼저 입력 받은 정점의 시선 벡터, 법선 벡터를 다시 정규화 합니다. 이것은 이들 값들이 텍스처 좌표로 넘어와서 픽셀 처리기가 보간 하기 때문에 단위 벡터인 크기를 다시 1로 조정하기 위해서입니다. 그리고 픽셀 쉐이더 버전은 반드시 2.0 이상으로 설정해야 제대로 동작합니다.

```

// 픽셀 처리 프로세스: 반드시 2.0 이상 버전 필요
float4 PxlPrc(SVsOut In) : COLOR
...
    float3 E = normalize(In.Eye);    // 시선 벡터 정규화
    float3 N = normalize(In.Nor);    // 법선 벡터 정규화
    float3 L = normalize(-m_vcLgt);
    float3 R = reflect(-L, N);      // 반사 벡터: 내장 함수 reflect() 이용. 부호 주의
    float4 S = pow( max(0, dot(R, E)), m_fShrp);    // 끝 반사

```

전체 코드는 [ht12\\_spec2\\_pxl.zip](#)에 구현 되어 있으며 실행하면 다음과 같은 장면을 볼 수 있습니다.



<픽셀 쉐이더 2.0 이상을 사용한 Specular 효과: [ht12\\_spec1\\_pxl.zip](#)>

분산 조명은 텍스처의 색상과 곱셈 연산을 했지만 Specular 효과는 빛의 고 휘도 부분을 표현한 것이므로 곱셈 연산 대신 덧셈을 합니다. 이것을 HLSL로 다음과 같이 구현합니다.

```

texture m_TxDif;
sampler SampDif = sampler_state

```

```

{
    texture = (m_TxDif);
...
float4 PxlPrc(SVsOut In) : COLOR
{
    float4 Out;                                // 출력 색상
...
    float4 S = pow( max(0, dot(R, E)), m_fShrp); // 풍 반사
...
    Out = tex2D( SampDiff, In.Tex );           // 텍스처 색상 추출
    Out += S;                                  // 풍 반사 값과 더함
...
    return Out;
}

```



<풍 반사와 텍스처 혼합: [ht12\\_spec3\\_tex.zip](#)>

D3D의 조명 효과를 정리하는 의미로 Specular 효과와 분산 조명에 대한 다중 조명을 구현해 봅시다. 먼저 풍 반사로 구현된 Specular 색상을 수식으로 표현해 봅시다.

$$\text{Specular 색상} = \sum \text{dot}(E, R_i)^{\text{Sharpness}} * \text{Lighting Color}_i$$

만약 Blinn-Phong 반사라면 시선 벡터  $E$ 와 빛의 반사 벡터  $R$  대신 Half 벡터  $H$ 와 법선 벡터  $N$ 을 사용합니다. 이 수식에 이전의 분산 조명 수식을 더하면 조명 효과에 대한 Specular 반사가 완성됩니다.

$$\text{최종 색상} = \sum (\text{dot}(N, L_i + \text{dot}(E, R_i)^{\text{Sharpness}}) * \text{Lighting Color}_i)$$

그런데 텍스처가 적용되면 분산 조명은 텍스처 색상과 곱셈을 하고 Specular를 더하는 방법을 가장 많이 사용합니다.

$$\begin{aligned}\text{최종 색상} &= \sum \text{Texture} * \text{dot}(N, L_i) * \text{Diffuse LightingColor}_i \\ &\quad + \sum \text{dot}(E, R_i)^{\text{Sharpness}} * \text{Specular Lighting Color}_i\end{aligned}$$

조금 긴 수식이지만 구현하는데 어려움이 없습니다.

```
float m_fShrp;           // Sharpness
float3 m_LgtDir[4];      // 조명 방향 벡터 배열
float4 m_LgtDiff[4];     // 조명 색상 배열
...
float4 PxlPrc(SVsOut In) : COLOR0
...
float3 N = normalize(In.Nor); // 법선 벡터 정규화
float3 E = normalize(In.Eye); // 시선 벡터 정규화

float3 L[3]; float3 R[3]; float4 Lam = 0; float4 Spc = 0;

for(i=0; i<3; ++i)          // 반사 벡터 계산
{
    L[i] = normalize(-m_LgtDir[i]);
    R[i] = reflect(-L[i],N);
}

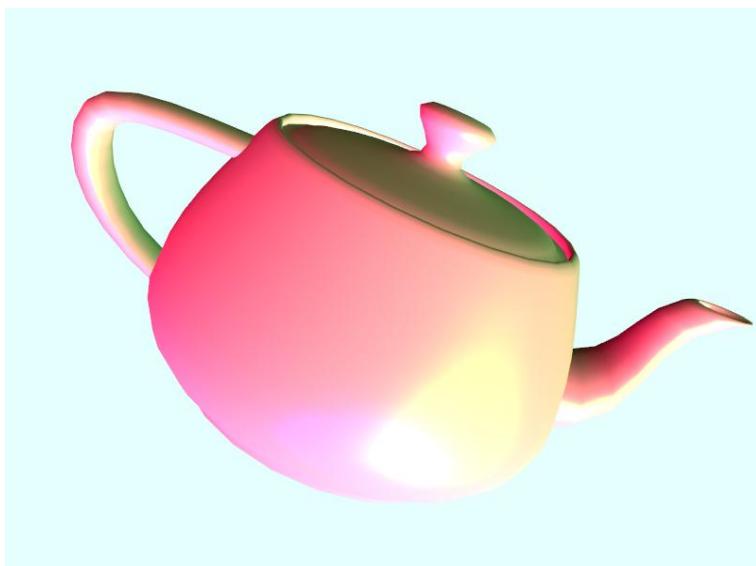
for(i=0; i<3; ++i)          // 분산 조명: Σdot(N, L_i) * Color_i
    Lam += (0.5f * dot(N, L[i]) + 0.5)*m_LgtDiff[i];

for(i=0; i<3; ++i)          // 풍 반사: Σdot(E, R_i)^Sharpness * Color_i
    Spc += pow( max(0, dot(E, R[i])), m_fShrp)*m_LgtDiff[i];
```

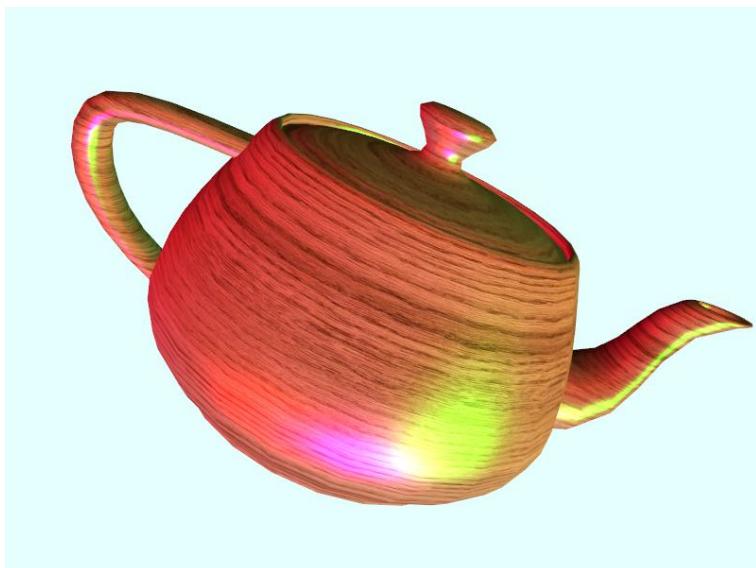
```
Out = saturate(Lam);           // 색상의 범위 [0,1]로 제한

if(1 == m_bTexture)           // 분산 조명 값과 텍스처 혼합
    Out *= tex2D( SampDif, In.Tex );

Out += Spc;                   // Specular 값을 더함
...
```



<다중 Specular와 분산 조명: [ht12\\_spec4\\_multi.zip](#) - R 키>



<다중 Specular와 분산 조명: [ht12\\_spec4\\_multi.zip](#) - T 키>

for 문에서 3개의 조명만 사용한 것은 ps\_2\_0의 명령 슬롯이 적기 때문이며 ps\_3\_0이상 지원이 되는 GPU는 보다 많은 명령 슬롯을 가지고 있어서 이것을 가지고 있는 분들은 조명의 숫자를 좀 더 늘려서 연습하기 바랍니다.

## 5.5 NPR(Non-photorealistic rendering) for Lighting

### 5.5.1 Cartoon Shading

우리는 저 수준 쉐이더 강의에서 툰(Toon: Cartoon) 쉐이딩 방법을 구현해 보았습니다. 툰 쉐이딩을 간단히 정리하면 연속적인 반사의 밝기를 이산적인(Discrete) 밝기로 만드는 것입니다. 이를 구현하기 위해서 이산적인 값은 불연속적인 텍스처의 흑백 색상으로 만들고 조명의 밝기를 이 텍스처의 좌표로 만들어 샘플링을 통해서 불연속 텍스처의 색상을 가져오도록 하면 툰 쉐이딩 구현이 완료 됩니다.

저 수준에서 해보았던 툰 쉐이딩을 HLSL로 만들어 봅시다. 먼저 정점 출력 구조체에 조명의 밝기를 저장할 수 있도록 1차원 텍스처 좌표를 Semantic으로 하는 구조체를 구성합니다.

```
struct SVsOut
{
    float4 Pos : POSITION;           // 출력 위치
    float Toon: TEXCOORD7;          // 1차원 Toon Texture 좌표
};
```

정점 처리에서는 Lambert 공식으로 조명의 반사 세기(Intensity)를 계산하고 이 반사 값을 1차원 텍스처 좌표에 저장합니다.

```
float3x3      m_mtRot;           // 회전 행렬
float3        m_vcLgt;            // 조명의 방향 벡터
...
SVsOut VtxPrc( ..., float3 Nor : NORMAL           // 입력 법선 벡터
...
float3 N = normalize(mul(Nor, m_mtRot));           // 법선 벡터의 회전
float3 L = normalize(-m_vcLgt);
float4 D = 0.4 + 0.6 * dot(N, L);                 // 조명의 밝기 계산
...
// [0, 1]로 제한된 밝기를 1차원 텍스처 좌표로 저장
```

```
Out.Toon = saturate(D);
return Out;
```

툰 효과를 만들기 위해서 다음과 같은 텍스처를 생성하고 이것을 샘플링 할 수 있도록 샘플러를 만듭니다.

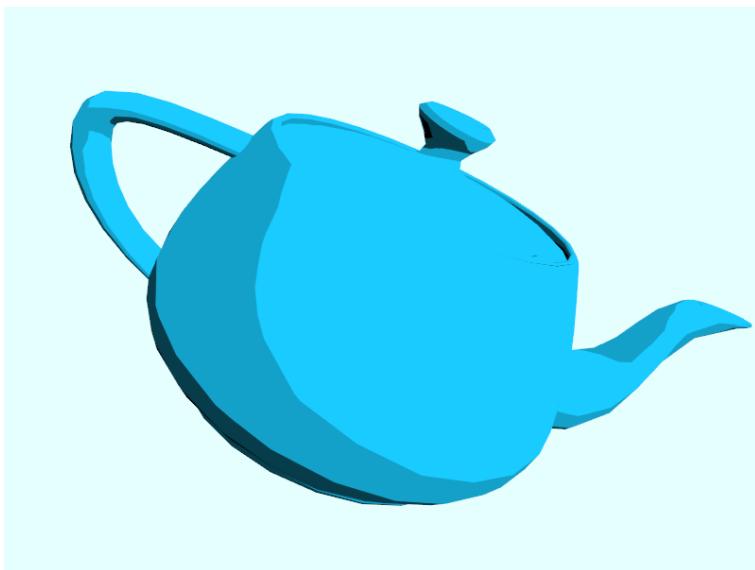


<카툰(Cartoon) 텍스처>

```
texture m_TxToon;           // Toon 텍스처
sampler SampToon = sampler_state // 샘플러
{
    texture = (m_TxToon);
...
};
```

정점 처리 과정에서 만들어진 1차원 텍스처 좌표를 픽셀 처리 과정에서 샘플링하고 외부에서 주어진 색상과 곱하면 툰 쉐이динг이 완성 됩니다.

```
float4 m_ToonColor;      // Toon 색상
...
float4 Px1Prc(SVsOut In) : COLOR0
{
    float4 Out;
    Out = tex2D( SampToon, In.Toon ) * m_ToonColor;
    return Out;
}
```



<Toon Shading: [ht13\\_toon1\\_basic.zip](#)>

툰 쇼이딩에 종종 외곽선을 적용하기도 합니다. 외곽선을 적용하는 가장 쉬운 방법은 법선 벡터를 정점의 위치에 더하고 이것을 CW로 그리는 것입니다. 이 방법은 Glow 효과 때도 구현해 보았으므로 이것을 그대로 사용하면 됩니다.

```
// 외곽선용 정점 처리 프로세스
SVsOut VtxPrc0(float4 Pos : POSITION0    // 입력 정점 위치
                , float4 Nor : NORMAL0    // 정점 법선 벡터
...
float4 P = Pos;
float4 N = Nor;

P += N * .05F;           // 정점의 위치에 법선 벡터를 더한다.
P.w = 1.0f;              // w=1로 한다.
P = mul(P, m_mtWld);   // 월드 변환
...
Out.Pos = P;            // 출력 위치에 복사
...
```

픽셀 처리 함수는 외곽선 처리는 외부에서 주어진 색상으로 단색을 만들고 툰 쇼이딩은 툰 텍스처에서 샘플링 할 수 있도록 합니다.

```
float4 PxlPrc(SVsOut In, uniform int bTexture) : COLOR0
...
...
```

```

if(0==bTexture)           // 외곽선 처리
    Out = In.Dff;
else                      // 투 효과 처리
    Out = tex2D( SampToon, In.Toon );

return Out;

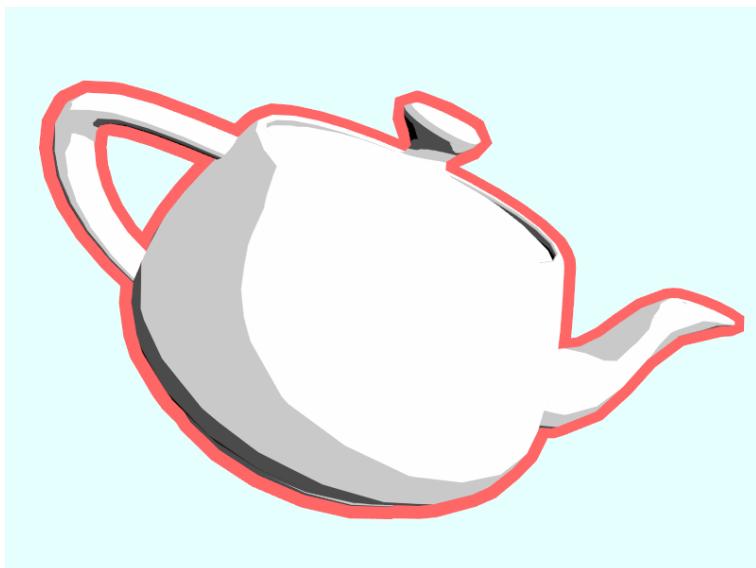
```

technique에서 외곽선을 CW로 먼저 그릴 수 있도록 하기 위해서 pass 0에 정점 쉐이더와 픽셀 쉐이더 객체를 구성 합니다. pass 1에서는 투 쉐이딩이 적용되도록 CCW로 Cull Mode를 구성합니다.

```

technique Tech0
{
    pass P0
    {
        CULLMODE      = CW;
        ZWRITEENABLE = FALSE;
        VertexShader = compile vs_2_0 VtxPrc0();
        PixelShader   = compile ps_2_0 PxlPrc(0);
    }
    pass P1
    {
        CULLMODE      = CCW;
        ZWRITEENABLE = TRUE;
        VertexShader = compile vs_2_0 VtxPrc1();
        PixelShader   = compile ps_2_0 PxlPrc(1);
    }
}

```



<툰 쉐이딩 + 외곽선: [ht13\\_toon2\\_edge.zip](#)>

카툰 효과는 분산 조명에 대한 이산 효과라 한다면 Diffuse 텍스처와 함께 렌더링 물체에 적용할 때 분산 조명과 마찬가지로 이들을 곱해야 합니다. 그런데 단순하게 곱해 버리면 카툰 효과를 충분히 발휘 할 수 없을 수도 있습니다. 카툰 효과는 주어진 툰 텍스처 보다 밝기(Intensity)뿐만 아니라 Contrast를 높여야 할 때도 있습니다. 밝기를 올리려면 툰 효과에 사용되는 밝고 어두운 텍스처에 적절한 값을 곱하고 Contrast는 쉐이더의 pow() 함수를 사용합니다.

```
float4 Px1Prc(SVsOut In) : COLOR
```

```
...
```

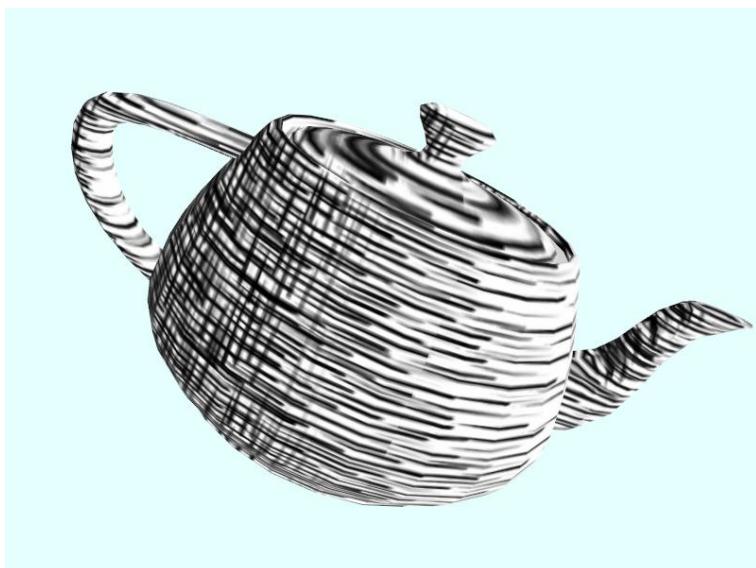
```
    Out = tex2D(SampToon, In.Toon)*1.2; // 적당한 값을 곱해서 밝기를 올림
    Out = pow(Out, 3)/1.2;           // pow() 함수 이용, contrast를 높임
    Out *= tex2D( SampDiff, In.Tx0 ); // Diffuse 텍스처와 혼합
    Out.a = 1;
    return Out;
```



<툰+텍스처: [ht13\\_toon3\\_diffuse.zip](#)>

### 5.5.2 Hatching

3D의 조명의 원리를 이해하고 있다면 툰 쉐이딩을 쉽게 구현 할 수 있음을 우리는 잘 알 수 있습니다. 툰 쉐이딩 이외에 조명 원리를 간단하게 적용해서 구현해 볼 수 있는 것이 Hatching입니다. [ht13\\_hatching\\_tonal\\_art1.zip](#) 예제를 실행하면 반사의 세기에 따라서 적당한 Hatching으로 만들어진 텍스처가 적용되고 있음을 볼 수 있습니다.



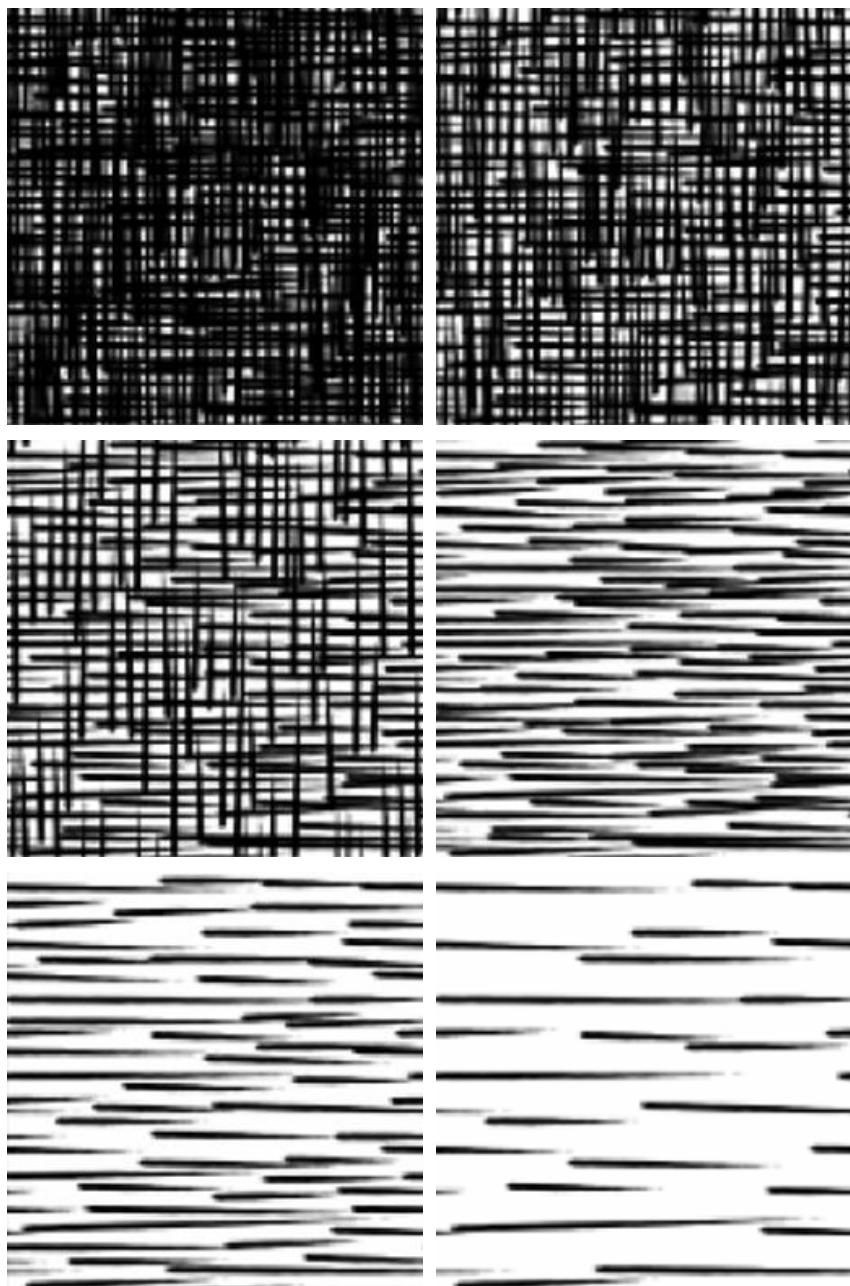
<NPR-tonal art map. [ht13\\_hatching\\_tonal\\_art1.zip](#)>

이 기술은 의외로 간단합니다. 구현 방법은 반사의 밝기를 일정한 구간으로 나누어서 구간에 해당하는 2개의 텍스처를 적절하게 혼합하는 것입니다. 예를 들어서 밝기에 대한 텍스처 A, B, C, D, E 5장이 준비되어 있다면  $[0., 0.25)$  범위는 A와 B 텍스처를,  $[0.25, 0.5)$  구간에 대해서는 B, C 를,  $[0.5, 0.75)$  C, D를, 그리고  $[0.75, 1.0)$  범위는 D, E 텍스처를 사용해서 혼합하는 것입니다.

그런데 이렇게 전체 밝기를  $[0, 1]$  사이로 하면 혼합하는 텍스처의 비중(Weight)를 구하는 것이 혼잡하기 때문에 각 구간의 크기를 1로 정하고, 반사의 밝기를 텍스처 개수( $n-1$ ) 만큼 곱합니다. 이렇게 하면 각각의 구간은  $[0., 1.)$ ,  $[1.0, 2.)$ , ...,  $[n-2, n-1)$ ,  $[n-1, \infty)$ 이 만들어 집니다. 만약 밝기가 0.7이고 전체 텍스처가 A(0), B(1), C(2), D(3), E(4) 5 장이면  $0.7 * (5-1) = 2.8$ 로 계산을 하고  $[2., 3.)$  구간의 텍스처 C와 D에 비중  $(2.8-2)$ 와  $(3 - 2.8)$ 을 각각 곱하고 곱한 결과를 더하면 최종 출력 색상이 됩니다.

간단한 원리 이기 때문에 곧바로 이것을 쉐이더로 구현해 봅시다. 만약 다음과 같이 밝기에 대한 6장의 텍스처가 준비되어 있다면 이들에 대한 샘플러 또한 6개를 선언해야 합니다.

```
sampler smp0 : register(s0) = sampler_state{...};  
sampler smp0 : register(s1) = sampler_state{...};  
sampler smp0 : register(s2) = sampler_state{...};  
sampler smp0 : register(s3) = sampler_state{...};  
sampler smp0 : register(s4) = sampler_state{...};  
sampler smp0 : register(s5) = sampler_state{...};
```



<밝기에 대한 Hatching Texture>

반사 모델은 Lambert 확산을 적용한다면 정점 처리 과정에서 반사의 세기를 계산해도 충분합니다.

```

struct SVsOut
{
    float4 Pos : POSITION;
    float4 Dif : COLOR0;           // 분산 조명 밝기
    float2 Tex : TEXCOORD0;        // 디퓨즈 맵 좌표
};

...
float3x3 m_mtRot;             // 회전 행렬
float3 m_vcLgt;               // 조명의 방향 벡터

SVsOut VtxPrc(..., float4 Nor : NORMAL0)
{
    SVsOut Out = (SVsOut)0;
    ...

    float3 N = normalize(mul(Nor,m_mtRot)); // 법선 벡터의 회전
    float3 L = normalize(m_vcLgt);
    float4 D = (1.f+dot(N, L)) * 0.5f;      // Lambert 확산으로 밝기 설정
    ...

    Out.Dif = D;                          // 출력 구조체에 복사
}

```

픽셀 처리 과정의 최종 색상은 각 구간에 맞게 비중(Weight)을 구하고 해당 텍스처에 곱한 다음 더하면 됩니다.

```

float4 Px1Prc(SVsOut In) : COLOR0
...
float htcLvl = In.Dif * 5.3; // 밝기를 5보다 약간 큰 5.3배를 한다

// Hatching 비중 값을 전부 0으로 설정
float Htch0 = 0;
float Htch1 = 0;
float Htch2 = 0;
float Htch3 = 0;
float Htch4 = 0;
float Htch5 = 0;

```

```

// 밝기에 대한 구간을 찾고 비중을 계산
if(htcLvl > 5.0)
{
    Htch0 = 1.0;
}

...
else if(htcLvl > 2.0)
{
    Htch2 = 1.0 - (3.0 - htcLvl);
    Htch3 = 1.0 - Htch2;
}
else if(htcLvl > 1.0)
{
    Htch3 = 1.0 - (2.0 - htcLvl);
    Htch4 = 1.0 - Htch3;
}
...

// 텍스처 샘플링 값에 해당 텍스처의 비중을 곱함
float4 t0 = tex2D(smp0, In.Tex*m_fHtchW) * Htch0;
float4 t1 = tex2D(smp1, In.Tex*m_fHtchW) * Htch1;
float4 t2 = tex2D(smp2, In.Tex*m_fHtchW) * Htch2;
float4 t3 = tex2D(smp3, In.Tex*m_fHtchW) * Htch3;
float4 t4 = tex2D(smp4, In.Tex*m_fHtchW) * Htch4;
float4 t5 = tex2D(smp5, In.Tex*m_fHtchW) * Htch5;

// 색상을 전부 더하고 출력
Out = t0 + t1 + t2 + t3 + t4 + t5;
Out.a = 1.0;
...

return Out;

```

만약 Diffuse 텍스처를 적용하려면 다음과 같이 마지막 단계에서 Diffuse 텍스처를 샘플링하고 출력 색상에 곱하면 됩니다.

```

Out = t0 + t1 + ... + t5;
Out.a = 1.0;
Out = pow(Out, 0.6)*1.5; // pow()함수로 Contrast를 낮춤

```

```

Out *= tex2D( SampDif, In.Tex );           // Multiply Diffuse Texture
return Out;

```



<Diffuse 텍스처가 적용된 NPR-tonal art map. [ht13\\_hatching\\_tonal\\_art2.zip](#)>

## 5.6 Diffuse and Lighting Mapping

어떤 오래된 책의 번역을 보면 텍스처를 질감으로 번역해 놓은 경우도 있습니다. 이론적으로 정점만 있어도 모든 가상의 물체를 표현할 수 있지만 현실과 같은 효과를 주기 위해서는 수백만 개 이상의 정점이 필요할지도 모릅니다. 텍스처는 화면에 동시에 연출 되는 정점 수에 대한 메모리를 줄이고 3차원 표면을 사실처럼 묘사하기 위해 2차원 또는 3 차원 이상의 픽셀로 구성되어 있는 것을 3D 기초와 지금까지의 강의 내용으로 잘 알고 있습니다.

텍스처를 정점에 적용하는 것을 매핑이라 하는데 매핑은 단순하게 정점에 2차원(혹은 1차원, 3차원) 좌표를 설정하는 것으로 그래픽 카드는 이것을 픽셀 처리 과정에서 해당 좌표에서 적당한 샘플링 방법으로 색을 추출한 다음 정점의 색상과 혼합해서 최종 색상을 만들어냅니다.

매핑은 그 목적과 구현 방법에 따라서 여러 종류가 있고 게임에서는 Diffuse Mapping, Lighting Mapping, Bump Mapping, Specular Mapping, Parallax Mapping, Environment Mapping, Shadow Mapping, Displacement Mapping 등이 있습니다. 또한 이들 매핑에서 적용되는 텍스처들은 00 Map이라 부릅니다. 예를 들어서 디퓨즈 매핑에 사용되는 텍스처를 디퓨즈 맵이라 하고, 라이팅 매핑

에 사용되는 텍스처를 라이팅 맵, 범프 매핑에 적용되는 텍스처를 법선 맵(Normal Map) 또는 범프 맵(Bump Map), 그리고 스페큘러 매핑에 사용되는 텍스처를 스페큘러 맵이라 합니다.  
이 강의에서는 게임에서 적용되는 각각의 매핑 기술의 원리와 구현을 살펴보겠습니다.

디퓨즈 매핑(Diffuse Mapping)은 우리가 가장 기초적으로 사용하는 매핑 기술로 정점에 단순히 색상을 적용하는 것으로 물체의 표면에 반사되는 난반사에 대한 분산 광 효과를 표현하기 때문에 디퓨즈 매핑이라 합니다. 라이팅 매핑(Lighting Mapping)은 그림자 또는 조명의 밝은 부분 표현하기 위해서 밝기(음영)에 대한 값을 텍스처(라이팅 맵)로 만들고 디퓨즈 맵과 Multi-Texturing으로 처리한 픽셀 처리에서 혼합 다중 텍스처 처리로 처리합니다.

3D 게임 초창기 그래픽 카드는 화면에 연출되는 물체의 정점 수가 그리 많지 않았을 때 광원에 대한 효과도 제대로 살리기 어려웠는데 이것을 텍스처로 해결했습니다. 즉, 정점에 텍스처에 빛의 광원 효과를 저장해서 장면 연출을 할 때 디퓨즈 맵과 다중 텍스처 처리를 합니다.

라이팅 매핑은 부드러운 조명 효과를 적절하게 만들어 주고 표현 또한 우수해서 ID 회사의 게임 '3D 퀘이크'에서 첫 선을 보이면서 현재까지 광범위하게 사용되고 있는 기술이며 아직까지 대부분의 3D 엔진이나 지형 툴에서 라이팅 맵 설정을 지원하고 있으며 실제 게임에서도 많이 응용되고 있습니다.

현재 일부 엔진의 경우 장면의 렌더링 속도를 위해서 지형의 경우 라이팅 맵과 디퓨즈 맵을 통합해서 하나의 디퓨즈 맵으로 가져 가는 경우도 있습니다.

라이팅 매핑은 다음과 같이 두 장의 텍스처를 Modulate 연산으로 처리합니다.

최종 색상 = Diffuse Map \* Lighting Map \* 밝기 상수

이 연산 방법은 3D 기초 시간의 다중 텍스처 처리(Multi-Texturing)에서 연습한 내용이고 구현하기가 너무나 간단해서 쉐이더를 사용하지 않고 고정 기능 파이프라인에서도 얼마든지 처리할 수 있습니다.

고정 기능 파이프 라인에서 이것을 처리할 때도 앞서 배운 Effect의 Technique에서 상태 설정을 하게 되면 좀 더 간결해지고 쉬어집니다.

```
// 디퓨즈 맵
texture m_TxDif;
sampler SampDif = sampler_state { texture = <m_TxDif>; ... };

// 라이팅 맵
texture m_TxLgt;
```

```

sampler SampLgt = sampler_state { texture = <m_TxLgt>; ... };

technique Tech0
...
pass P1
{
    sampler [0] = (SampDif);
    sampler [1] = (SampLgt);

    COLORARG1[0] = TEXTURE;
    COLORARG2[0] = DIFFUSE;
    COLOROP [0] = SELECTARG1;
    ALPHAOP [0] = DISABLE;
    ALPHAARG1[0] = TEXTURE;
    ALPHAOP [0] = SELECTARG1;
    ...
    TEXCOORDINDEX[1] = 0;           // 0 단계 텍스처 좌표 사용
    COLORARG1[1] = TEXTURE;
    COLORARG2[1] = CURRENT;
    COLOROP [1] = MODULATE2X;      // 색상 연산
    ALPHAARG1[1] = TEXTURE;
    ALPHAARG2[1] = CURRENT;
    ALPHAOP [1] = MODULATE;

    COLOROP [2] = DISABLE;
    ALPHAOP [2] = DISABLE;
    ...
}

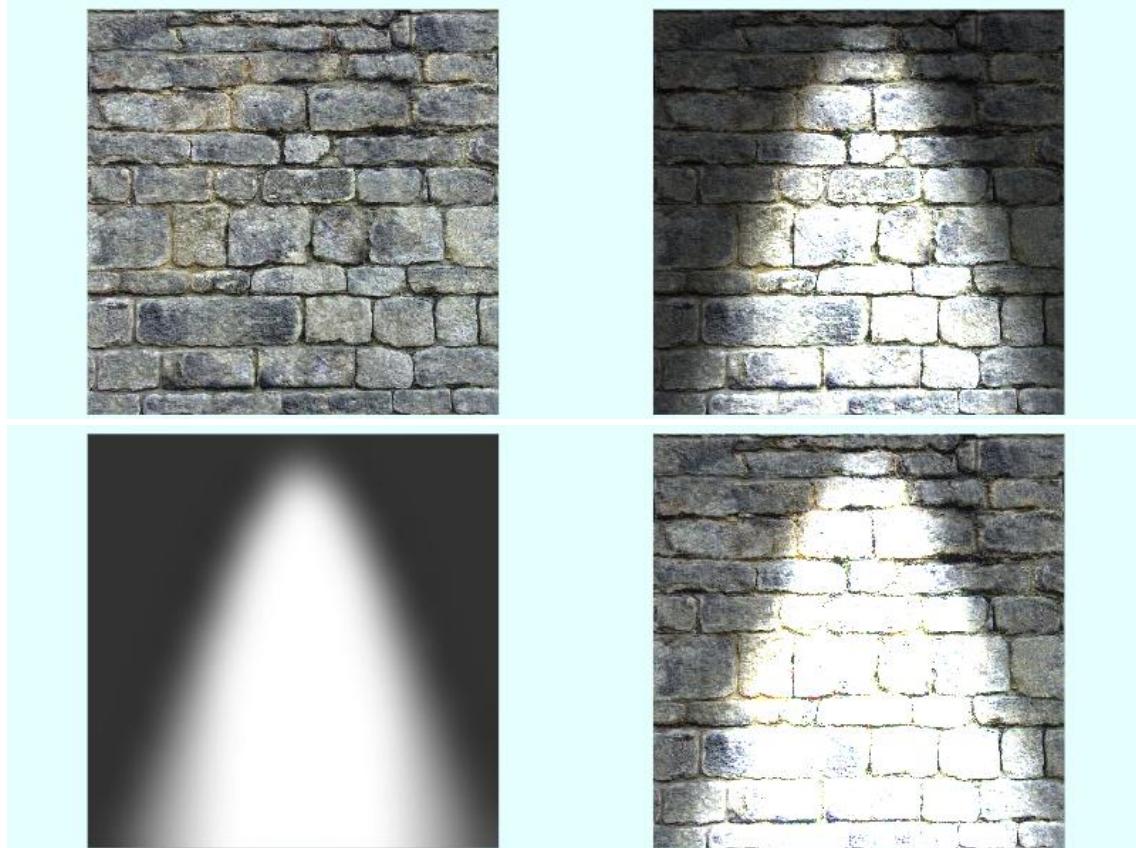
```

다중 텍스처 처리 0 단계에서 색상의 입력 값 1번((Color Argument1: COLORARG1[0]))을 텍스처로 설정하고 정점의 색상 값을 입력 값 2번((Color Argument1: COLORARG2[0]))으로 설정합니다. 색상 혼합은 텍스처를 선택합니다. 텍스처의 색상과 정점의 색상을 혼합하는 OP 값을 MODULATE하는 것이 보통이지만 현재는 순전히 텍스처의 색상만 적용하므로 OP 값을 stage 입력의 1번을 선택 합니다. 알파는 텍스처의 알파를 그대로 사용합니다.

다중 텍스처 처리 1 단계에서는 0 단계에서 출력한 색상 Current를 가지고 라이팅 맵과 혼합하는 단계입니다. 입력 색상 1번(COLORARG1[1])은 라이팅 맵의 텍스처를 선택하고, 0 단계의 출력 값 CURRENT는 COLORARG2로 지정해서 이 둘을 곱(MODULATE) 하도록 합니다. 만약 색상에 대한 OP를 MODULATE로 선택했을 때 어둡게 나오면 MODULATE2X나 MODULATE4X를 선택합니다.

알파 값은 두 텍스처의 곱셈인 MODULATE를 선택합니다. 다음 단계에서 다중 텍스처 처리를 안 하도록 COLOROP[2]와 ALPHAOP[2] 를 DISABLE로 합니다.

다음의 [ht21\\_lightmap\\_fixed.zip](#) 예제는 색상의 혼합을 MODULATE2X와 MODULATE4X를 사용했습니다.



<고정 기능 파이프라인의 라이팅 매핑 . [ht21\\_lightmap\\_fixed.zip](#)>

고정 파이프라인에서 라이팅 맵을 사용하기 위해서 다중 텍스처 처리를 사용하기 위해서 각 단계에 대한 상태 값을 설정했지만 쉐이더를 사용하면 라이팅 맵과 디퓨즈 맵은 더하기, 빼기, 곱하기, 나누기 등의 적당한 산술 식으로 라이팅 매핑을 다음과 구현 할 수 있습니다.

```
// 라이팅 매핑 픽셀 처리 함수
float4 Px1Prc(SVsOut In, uniform int nIntensity) : COLOR
{
    float4 Out;
    float4 D = tex2D( SampDif, In.Tx0 );           // Diffuse Map
    float4 L = tex2D( SampLgt, In.Tx0 );           // Lighting Map

    Out = D;
    if(0!=nIntensity)
```

```

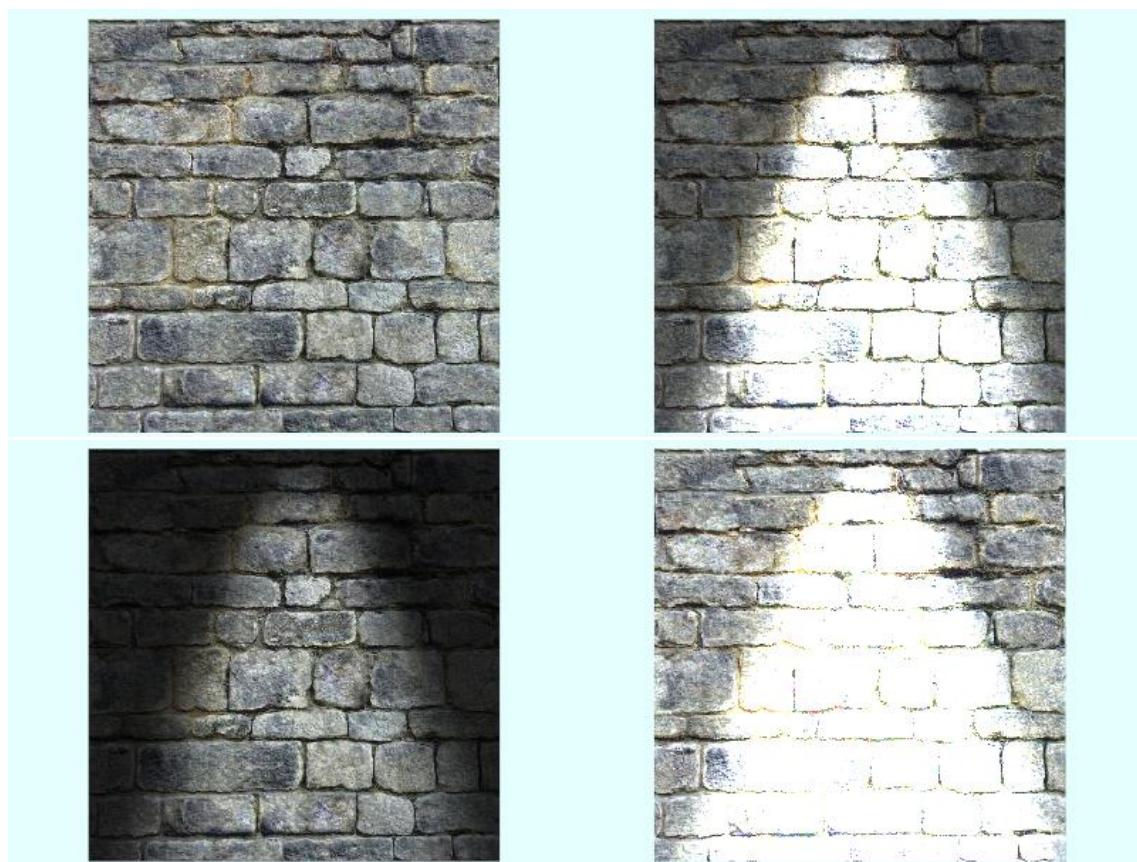
    Out *= L * nIntensity;

    return Out;
}

```

HLSL을 사용하면 좀 더 공식에 맞는 코드를 구현할 수 있습니다. 또한 내장 함수 pow() 등을 사용해 Contrast도 적절하게 만들어 낼 수 있습니다.

[ht21\\_lightmap\\_shader.zip](#)은 라이팅 매핑의 공식을 그대로 사용한 예제로 전체 밝기를 x1, x3, x6을 구현했습니다.



<Programmable Pipeline Lighting Mapping. [ht21\\_lightmap\\_shader.zip](#)>

## 5.7 Bump Mapping (Normal Mapping)

범프 매핑(Bump Mapping)은 적은 수의 정점을 가지고 보다 높은 광원 효과를 만들기 위해서 법선 벡터를 텍스처에 저장한 법선 맵(Normal Map)을 픽셀 처리에서 픽셀 단위로 조명에 대한 반사 효과를 적용하는 방법입니다.

블린(Jim Blinn)은 정점의 숫자가 적은 단조로운 물체에 텍스처를 이용해서 밝고 어둠에 대한 음

영을 더 사실감 있는 효과를 내기 위해서 반사의 세기를 결정하기 위한 법선 벡터를 이미지에 저장하고 프로그램에서 이 이미지의 픽셀을 법선 벡터로 변환해서 분산 조명, Specular 조명을 계산하도록 했습니다.

법선 맵을 이용한 반사 효과는 정점의 개수에 거의 독립적이며 올록볼록한 Embossing 효과를 만들기 때문에 범프 매핑(Bump Mapping)이라 부르게 되었습니다.

정점의 법선 벡터를 이미지에 저장하기 위해서 그래픽 담당자들은 렌더링에 필요한 정점 수보다 훨씬 많은 고 폴리곤(High Polygon)으로 메쉬 작업을 먼저 합니다. 그 다음으로 그래픽 툴을 이용해서 법선 벡터를 이미지에 저장을 하고 다시 이 이미지를 재 작업을 합니다. 프로그래머는 간단하게 구현할 수 있지만 그래픽을 만드는 분들에게는 상당히 노동력이 많이 드는 일이 아닐 수 없습니다.

정점의 법선 벡터 대신에 법선 벡터를 저장한 법선 맵(Normal Map)에서 벡터를 가져오는 일만 다를 뿐 반사의 밝기를 정하는 기본 처리는 분산 조명 또는 Specular 조명에서 사용한 공식 그대로 적용이 됩니다.

D3D의 고정 기능 파이프라인은 법선 맵에서 법선 벡터를 추출하고 이 법선 벡터와 빛의 방향 벡터를 내적(dot product) 연산 방법을 제공하고 있습니다. 구현 방법은 먼저 [-1, 1] 범위의 빛의 방향 벡터를 [0, 255] 색상 범위 값으로 만들고 32 비트 DWORD형으로 Tfactor에 저장합니다. 그 다음으로 다중 텍스처 처리에서 색상 혼합 COLORROP를 DOTPRODUCT3로 합니다. DOTPRODUCT3는 Tfactor와 텍스처의 색상을 내적으로 흑백의 명암을 만들어냅니다.

프로그램의 구현은 먼저 빛의 방향 벡터를 [0, 255] 범위의 DWORD형으로 만드는 함수부터 작성합니다.

```
DWORD VectorToRGB(D3DXVECTOR3* vcNor)
{
    DWORD dwR = (DWORD)(127 * vcNor->x + 128);
    DWORD dwG = (DWORD)(127 * vcNor->y + 128);
    DWORD dwB = (DWORD)(127 * vcNor->z + 128);
    return (DWORD)(0xff000000 + (dwR << 16) + (dwG << 8) + dwB);
}
```

렌더링 머신의 상태 설정에서 빛의 방향 벡터가 DWORD형으로 변환된 값을 Tfactor에 저장합니다.

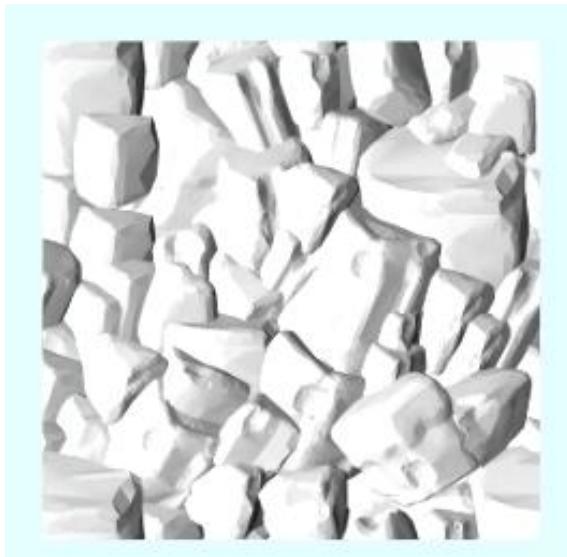
```
TEXTUREFACTOR= (m_dTFactor);
```

Color Arg1 또는 Arg2 둘 중 하나를 Tfactor로 지정하고, 법선 맵 텍스처는 나머지 Argument에 설정합니다.

```
COLORARG1[0] = TFACTOR;           // Tfactor 값을 ColorArg1으로 설정
COLORARG2[0] = TEXTURE;
```

Color Op를 DOTPRODUCT3로 설정하면 렌더링 머신이 Tfator와 텍스처의 픽셀을 내적(dot product) 연산을 합니다.

```
COLOROP [0] = DOTPRODUCT3;      // Tfator에 저장된 값과 텍스처의 픽셀을 내적
```



<고정 파이프라인에서 법선 맵과 T-factor를 사용한 밝기>

디퓨즈 맵과 혼합하려면 여러 단계의 다중 텍스처 처리가 필요합니다. 0 단계에서 디퓨즈 맵의 색상을 가져와 CURRENT에 저장합니다. 1 단계는 Tfator와 법선 맵의 색상을 내적을 하고 이 값을 TEMP에 저장합니다. 2 단계는 CURRENT에 저장된 값과 TEMP에 저장된 값을 혼합합니다.

```
sampler [0] = (SampDif);        // 디퓨즈 맵
sampler [1] = (SampNor);       // 법선 맵
```

```
TEXTUREFACTOR= (m_dTFactor);   // Tfactor
```

```
// 디퓨즈 맵의 색상 가져와서 CURRENT에 저장
COLORARG1[0] = TEXTURE;
COLOROP [0] = SELECTARG1;
RESULTARG[0] = CURRENT;
...
```

```
// Tfactor와 범선 맵의 색상을 내적 하고 결과를 TEMP에 저장
COLORARG1[1] = TFACTOR;
COLORARG2[1] = TEXTURE;
COLOROP [1] = DOTPRODUCT3;
RESULTARG[1] = TEMP;
...

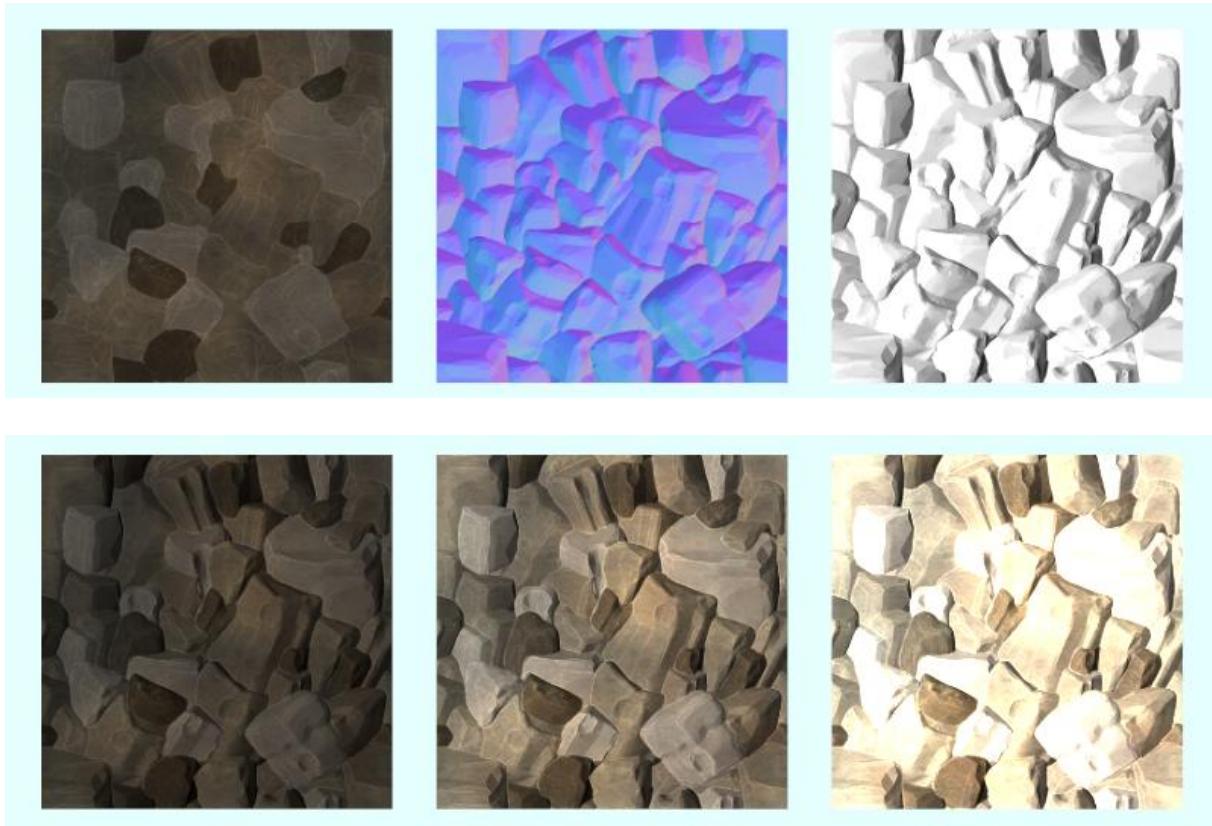
// Current와 Temp에 저장된 색상을 혼합
COLORARG1[2] = CURRENT;
COLORARG2[2] = TEMP;
COLOROP [2] = MODULATE;
RESULTARG[2] = CURRENT;
...
```

이 방법은 처리에 대해서 총 3 단계가 필요합니다. 그런데 Tfactor와 범선 맵의 연산을 먼저 진행하면 단계를 총 2 단계로 줄일 수 있습니다.

```
// 0 단계에서 Tfactor와 범선 맵을 dot 연산하고 결과를 CURRENT에 저장
COLORARG1[0] = TFACTOR;
COLORARG2[0] = TEXTURE;
COLOROP [0] = DOTPRODUCT3;
ALPHAOP [0] = DISABLE;
RESULTARG[0] = CURRENT;
...

// CURRENT의 저장되어 있는 색과 디퓨즈 맵을 혼합
COLORARG1[1] = CURRENT;
COLORARG2[1] = TEXTURE;
COLOROP [1] = MODULATE2X;
COLOROP [2] = DISABLE;
...
```

[ht22\\_bump\\_fixed.zip](#)는 Tfactor와 범선 맵의 내적 연산과 디퓨즈 맵과 MODULATE, MODULATE2X, MODULATE4X 혼합을 보여주고 있습니다.

<Fixed Function Pipeline Normal Mapping. [ht22\\_bump\\_fixed.zip](#)>

범프 효과를 고정 기능 파이프라인에서 간단하게 구현할 수 있지만 문제는 정점의 법선 벡터와 관련 없이 오직 빛의 방향만으로 설정된다는 것입니다. 과거 쉐이더가 지원 되지 않을 때 이 문제를 해결하기 위해서 다음 공식을 이용했습니다.

반사의 밝기 = `dot(정점의 법선 벡터, 빛의 방향 벡터)`

\* 반사에 대한 법선 벡터 연산 결과

이것을 구현하려면 제일 처음 제시했던 방법을 수정해서 Color Op를 텍스처만 선택되지 않고 Diffuse와 Modulate할 수 있게 다음과 같이 수정해야 합니다.

```
COLORARG1[0] = TEXTURE;
COLORARG2[0] = DIFFUSE;
COLOROP [0] = MODULATE;
RESULTARG[0] = CURRENT;
```

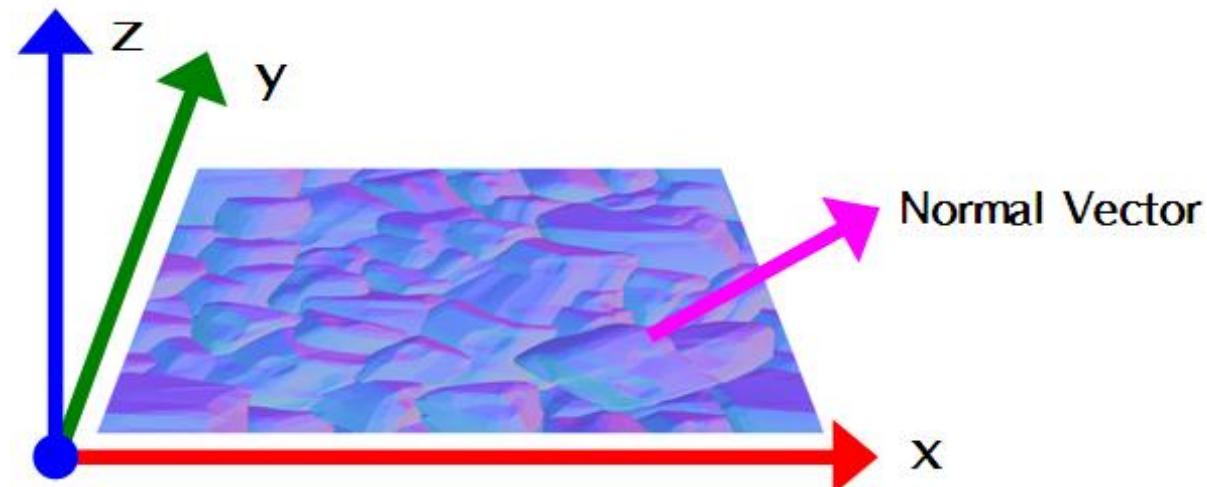
이것은 조명이 활성화 되어 있다면 정점 처리 과정의 Diffuse 색상은 조명과 정점의 법선 벡터 연산의 결과를 저장하고 있기 때문입니다.

다음으로 쉐이더를 사용한 범프 효과를 구현해 보겠습니다. 쉐이더를 사용하면 정점의 법선 벡터를 반영해서 좀 더 정교한 범프 효과를 구현 할 수 있습니다.

텍스처의 색상을 법선 벡터로 바꾸기 위해서 먼저 색상의 범위가 [0, 255]가 아닌 [0,1] 범위라는 것을 명심해야 합니다. 따라서 이것을 조명에 필요한 법선 벡터의 범위 [-1, 1]로 만들기 위해서 텍스처에서 얻은 색상에 2배를 하고 (-1)을 더해야 합니다.

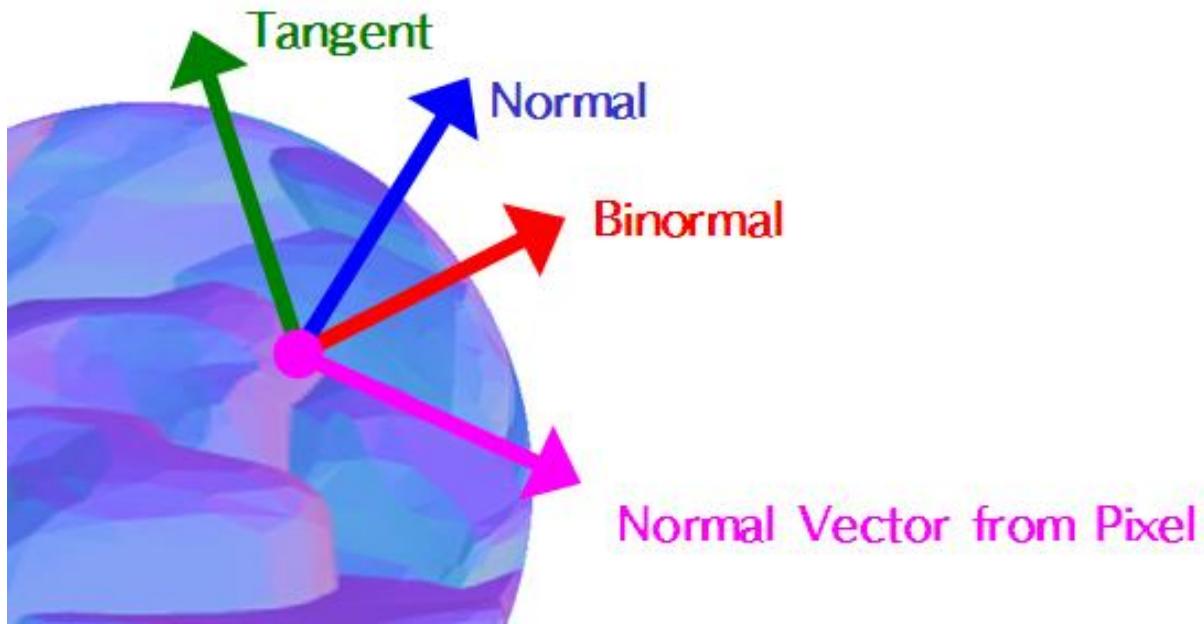
텍스처의 법선 벡터 =  $2 * \text{tex2D}(\text{법선 맵 샘플러}, \text{텍스처 좌표}).xyz - 1.0$

앞서 법선 맵은 고 폴리곤 작업 등을 통해서 제작이 된다고 했습니다. 그래픽 담당자들은 다음 그림처럼 x, y 평면에 z 방향이 위를 향하는 공간에 법선 벡터를 색상으로 저장합니다. 이렇게 Z축이 D3D의 Y축 방향과 일치하게 되는 이유는 법선 맵 텍스처 작업은 Max와 같은 그래픽 툴을 이용하며 대부분의 그래픽 툴은 x, y 평면에 z축이 위를 향하는 오른손 좌표계로 구성되어 있기 때문입니다.



<법선 맵 공간과 법선 벡터>

그리고 이 법선 맵은 다음 그림처럼 정점에 매핑이 되면 법선 맵 텍스처 공간의 z 방향은 정점의 법선 벡터(Normal Vector)에, 텍스처 공간의 y 방향 벡터는 정점의 접선 벡터(Tangent Vector)에, 그리고 x 방향 벡터는 종법선 벡터(Binormal Vector)에 대응 시키고 색상에서 추출한 법선 벡터를 이 세 개의 축인 정점의 법선, 접선, 종법선 벡터 공간으로 변환해야 합니다.



<색상에서 추출한 법선 벡터와 정점의 법선, 접선, 종법선 벡터로 구성된 공간>

법선, 접선, 종법선 벡터로 구성된 공간으로 텍스처에서 얻은 법선 벡터를 변환해야 하는데 이것을 쉽게 풀기 위해서 텍스처 공간의 x축(1, 0, 0), y축 (0, 1, 0), 그리고 z축 (0, 0, 1)은 어떤 변환 행렬  $M$ 에 의해 각각 종법선(Bx, By, Bz), 접선(Tx, Ty, Tz), 법선(Nx, Ny, Nz) 벡터로 만들어 진다고 합시다.

이것을 수식으로 표현 다음과 같습니다.

$$M \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix}, \quad M \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} Tx \\ Ty \\ Tz \end{pmatrix}, \quad M \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} Nx \\ Ny \\ Nz \end{pmatrix}$$

세 개의 축이 변환하는 것을 하나로 다음과 같이 합칠 수 있으며 이를 통해서 행렬  $M$ 을 아주 손쉽게 구할 수 있습니다.

$$M \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} Bx & Tx & Nx \\ By & Ty & Ny \\ Bz & Tz & Nz \end{pmatrix} = M \vec{I} = \begin{pmatrix} Bx & Tx & Nx \\ By & Ty & Ny \\ Bz & Tz & Nz \end{pmatrix}$$

$$M = \begin{pmatrix} Bx & Tx & Nx \\ By & Ty & Ny \\ Bz & Tz & Nz \end{pmatrix}$$

그런데 정점의 종법선, 접선, 법선 벡터들은 월드 행렬의 회전 변환을 할 수 있습니다. 따라서 회전 변환이 적용된  $M'$ 도 이와 같은 방법으로 구성합니다.

$$\begin{pmatrix} Bx' \\ By' \\ Bz' \end{pmatrix} = \vec{M}_{\text{rot}} \begin{pmatrix} Bx \\ By \\ Bz \end{pmatrix}, \quad \begin{pmatrix} Tx' \\ Ty' \\ Tz' \end{pmatrix} = \vec{M}_{\text{rot}} \begin{pmatrix} Tx \\ Ty \\ Tz \end{pmatrix}, \quad \begin{pmatrix} Nx' \\ Ny' \\ Nz' \end{pmatrix} = \vec{M}_{\text{rot}} \begin{pmatrix} Nx \\ Ny \\ Nz \end{pmatrix}$$

$$M' = \begin{pmatrix} Bx' & Tx' & Nx' \\ By' & Ty' & Ny' \\ Bz' & Tz' & Nz' \end{pmatrix}$$

이 회전 변환 행렬  $M'$ 에 색상에서 얻은 법선 벡터( $TxN.x$ ,  $TxN.y$ ,  $TxN.z$ )를 변환하면 되는데 이것은 행렬 \* 벡터 연산과 같은 의미입니다. 즉, 최종 법선 벡터는 " $M' * TxN$ "으로 만들어 집니다.

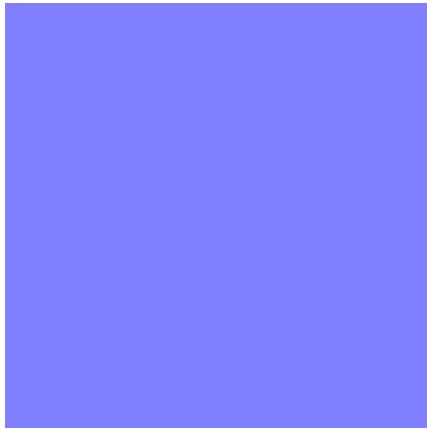
$$\begin{pmatrix} TxN'.x \\ TxN'.y \\ TxN'.z \end{pmatrix} = \begin{pmatrix} Bx' & Tx' & Nx' \\ By' & Ty' & Ny' \\ Bz' & Tz' & Nz' \end{pmatrix} \begin{pmatrix} TxN.x \\ TxN.y \\ TxN.z \end{pmatrix}$$

이것을 HLSL로 작성해야 하는데 퍽셀 처리에서는 행렬 연산이 불가능 할 수 있으므로 행렬 \* 벡터 연산을 풀어서 최종 법선 벡터를 만드는 것이 좋습니다.

$$\begin{aligned} TxN'.x &= B.x' * TxN.x + T.x' * TxN.y + N.x' * TxN.z \\ TxN'.y &= B.y' * TxN.x + T.y' * TxN.y + N.y' * TxN.z \\ TxN'.z &= B.z' * TxN.x + T.z' * TxN.y + N.z' * TxN.z \end{aligned}$$

이렇게 구한  $TxN'$ 을 분산 광원 효과, Specular 효과의 공식에서 사용되는 법선 벡터로 정해서 사용하면 됩니다.

지금까지의 내용이 맞는지 확인하기 위해서 다음과 같이 색상이 (127, 127, 255)로 구성된 텍스처를 준비합니다.



색상 값 (127, 127, 255)은 쉐이더에서 (0.5, 0.5, 1.0)이 되며 이 값에 2를 곱하고 1을 빼면 (0, 0, 1)이 됩니다. 이와 같은 텍스처를 구(Sphere) 형태의 물체에 매핑하고 조명을 적용했을 때 정점에 적용된 조명 효과와 같으면 범프 효과는 성공인 것이고 쉐이더 코드는 제대로 작성된 것이라 할 수 있습니다.

구체(Sphere)는 종법선, 접선, 법선 벡터를 갖고 있다면 다음과 같이 정점 구조체를 구성합니다.

```
struct VtxNUV1
{
    VEC3    p;        // Position
    VEC3    n;        // Normal
    FLOAT   u, v;    // Texture Coord
    VEC3    t;        // Tangent
    VEC3    b;        // Binormal
};
```

또한 정점 선언자는 이 구조체의 자료 순서에 맞게 작성합니다.

```
D3DVERTEXELEMENT9 dec1[] =
{
    {0, 0,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_POSITION,0},
    {0, 12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_NORMAL ,0},
    {0, 24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_TEXCOORD,0},
    {0, 32,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_TANGENT ,0},
    {0, 44,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_BINORMAL,0},
    D3DDECL_END()
```

```
};
```

이렇게 작성하는 것이 정석이지만 고정 기능 파이프라인의 FVF로 작성할 수 있습니다. 보통 텍스처 좌표는 하나의 정점에 하나 또는 2개 정도만 사용하는 것이 대부분이어서 사용할 수 있는 총 8개의 텍스처 좌표 중에서 남는 것을 3차원으로 하고 접선(Tangent), 종법선(Binormal)을 저장합니다.

```
enum
{
    FVF = D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_TEX3| \
        D3DFVF_TEXCOORDSIZE3(1)|D3DFVF_TEXCOORDSIZE3(2)
};
```

D3DFVF\_TEX3는 3개의 텍스처 좌표를 장을 사용하는 것을 의미하고 D3DFVF\_TEXCOORDSIZE3(1)는 1번째의 텍스처 좌표를 3차원으로 지정함을 의미합니다.

아직까지 쉐이더가 익숙하지 않은 분들은 FVF를 사용하는 것이 유리해 보입니다. 다음으로 HLSL를 작성합니다. 먼저 정점 처리의 출력 구조체에 종법선, 접선, 법선 벡터를 텍스처 좌표계에 저장될 수 있는 구조를 만듭니다.

```
// 종법선, 접선, 법선 벡터를 저장하기 위한 정점 출력 구조체
struct SVsOut
{
    float4 Pos : POSITION;
    float2 Tex : TEXCOORD0; // 디퓨즈 맵 텍스처 좌표
    float3 Bnr : TEXCOORD5; // 종법선 벡터
    float3 Tan : TEXCOORD6; // 접선 벡터
    float3 Nor : TEXCOORD7; // 법선 벡터
};
```

접선, 종법선 벡터를 텍스처 좌표를 사용했을 경우 입력 Semantic에 TEXCOORD#으로 지정합니다. DECLUSAGE를 사용했을 경우 다음의 접선 벡터에 대한 Semantic은 TANGENT, 종법선 벡터에 대한 Semantic은 BINORMAL을 사용합니다  
정점 처리 함수에서 접선, 종법선 벡터는 법선 벡터와 마찬가지로 회전 변환만 적용하고 이것을 정점 출력 구조체 변수에 저장합니다.

```
// 정점 처리 함수
```

```

SVsOut VtxPrc(float3 Pos : POSITION      // 입력 정점 위치 벡터
              , float4 Nor : NORMAL        // 법선 벡터
              , float2 Tex : TEXCOORD0    // 텍스처 좌표
              , float3 Tan : TEXCOORD1    // 접선 벡터. 텍스처 좌표 Semantic 사용
              , float3 Bnr : TEXCOORD2    // 종법선 벡터. 텍스처 좌표 Semantic 사용
...
float3 N = mul(Nor, m_mtRot);           // 법선 벡터의 회전
float3 T = mul(Tan, m_mtRot);          // 접선 벡터의 회전
float3 B = mul(Bnr, m_mtRot);          // 종법선 벡터의 회전
...
Out.Nor = N;                           // 회전 변환한 법선 벡터 저장
Out.Tan = T;                           // 회전 변환한 접선 벡터 저장
Out.Bnr = B;                           // 회전 변환한 종법선 벡터 저장
return Out;

```

정점 처리 함수는 법선, 접선, 종법선 벡터의 회전이 있을 뿐 다른 정점 처리 과정과 크게 다른 점은 없습니다.

픽셀 처리는 최소한 픽셀 쉐이더 2.0 이상 지원되는 GPU가 필요합니다. 이것은 픽셀 쉐이더 조명 때와 마찬가지로 정점의 접선, 종법선 벡터는 픽셀 처리기에 의해 선형 보간이 되고 이 보간된 벡터를 다시 단위 벡터로 만들어야 하는데 이 정도의 명령어 처리는 2.0이상 되어야 제대로 구현 되기 때문입니다.

픽셀 처리 함수는 먼저 입력 받은 법선, 접선, 종법선 벡터를 단위 벡터로 만듭니다.

```

// 픽셀 처리 함수
float4 PxlPrc(SVsOut In) : COLOR0
...
float3 B= normalize(In.Bnr);           // 입력 종법선 벡터를 단위 벡터로 만들
float3 T= normalize(In.Tan);          // 입력 접선 벡터를 단위 벡터로 만들
float3 N= normalize(In.Nor);          // 입력 법선 벡터를 단위 벡터로 만들
...

```

다음으로 법선 맵에서 색상을 추출해서 이 값을 [-1, 1] 범위의 벡터로 만듭니다. 이를 위해서 tex2D()함수로 추출한 색상 값에 2를 곱하고 1을 뺍니다.

```

float3 C1= 0;
float3 C = 2*tex2D(SampNor, In.Tex).xyz-1;
C = normalize(C);

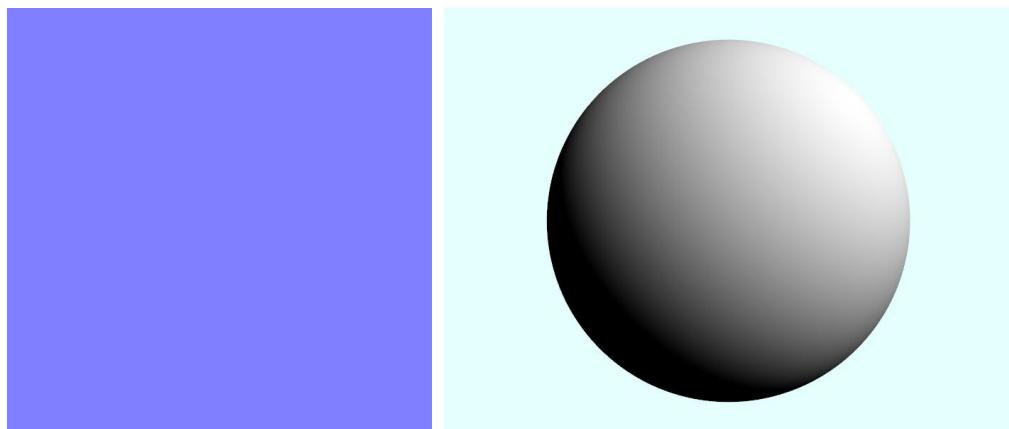
```

색상에서 추출한 이 벡터를 행렬의 변환과 같은 연산을 합니다.

```
C1.x = B.x * C.x + T.x * C.y + N.x * C.z;
C1.y = B.y * C.x + T.y * C.y + N.y * C.z;
C1.z = B.z * C.x + T.z * C.y + N.z * C.z;
```

마지막으로 dot() 함수 등을 사용해서 분산 조명 효과를 적용 합니다.

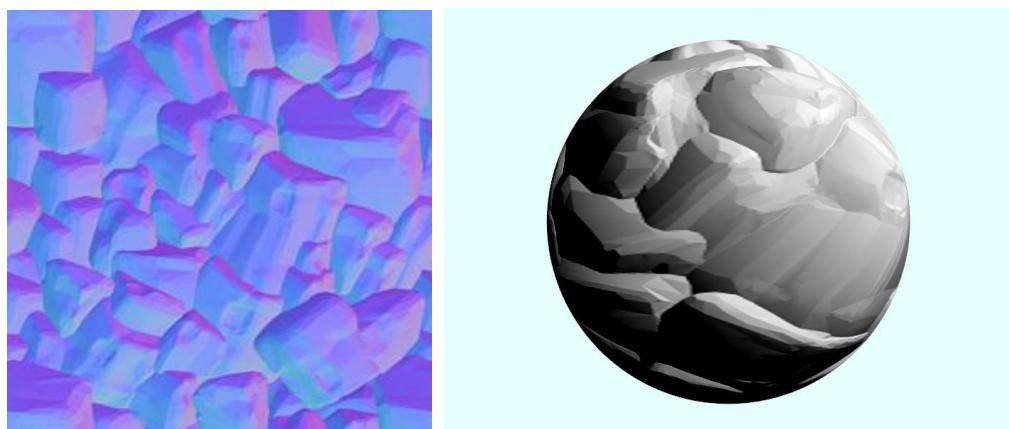
```
Bmp = dot(C1, Lgt);           // 분산 조명 적용
return Bmp;
```



<(0,0,1) 방향이 저장된 벡선 맵을 사용한 분산 조명 효과

: [ht22\\_bump1.zip](#), [ht22\\_bump2.zip](#), [ht22\\_bump3.zip](#)

(0, 0, 1) 방향으로 저장된 텍스처를 사용해서 원하는 형태의 조명 효과가 나왔으면 다음과 같은 벡선 맵을 적용해 봅니다.



<벡선 맵과 분산 조명 효과: [ht22\\_bump1.zip](#), [ht22\\_bump2.zip](#), [ht22\\_bump3.zip](#)

종법선 벡터는 정점의 법선 벡터와 접선 벡터의 외적으로 구할 수 있습니다.

```
종법선 벡터 = cross(법선 벡터, 접선 벡터)
```

이 원리를 이용하면 정점 구조체와 데이터의 크기는 작아집니다. 그런데 퍽셀 처리 함수는 종법선 벡터를 사용해야 하기 때문에 입력한 법선 벡터, 접선 벡터, 그리고 cross() 함수를 사용해서 종법선 벡터를 구합니다.

```
float3 T= normalize(In.Tan);
float3 N= normalize(In.Nor);
float3 B= normalize(cross(T, N));           // 외적으로 종법선 구함
```

이 방법을 사용하면 퍽셀 처리에서 cross() 함수를 사용하지만 정점의 데이터는 줄일 수 있으며 [ht22\\_bump2.zip](#)은 이것을 구현한 예제입니다.

모든 렌더링 물체가 접선 벡터 또는 종법선 벡터가 있으면 완전한 범프 효과를 만들 수 있지만 없는 경우에도 법선 벡터 만으로도 접선, 종법선 벡터를 외적을 이용해서 만들 수 있습니다. 예를 들어 데카르트 좌표계의 x축, y축, z축 방향 벡터는 서로 직각이며 이런 경우 x축 = cross(y축, z축), y축 = cross(z축, x축), z축 = cross(x축, y축)의 성질이 있습니다.

이것을 법선, 종법선, 접선에 적용하면 종 법선 벡터를 x축에 평행한 (1, 0, 0) 방향으로 정하고 접선 벡터를 법선 벡터와 (1, 0, 0)방향의 종법선 벡터의 외적으로 구하고 단위 벡터로 만듭니다. 다음으로 접선 벡터와 법선 벡터를 외적하고 이 벡터를 단위 벡터로 만들어 종법선 벡터로 설정합니다.

```
종법선 벡터 = float3(1, 0, 0)
접선 벡터 = normalize( cross(법선 벡터, 종법선 벡터) )
종법선 벡터 = normalize( cross(접선 벡터, 법선 벡터) )
```

HLSL로 작성하면 다음과 같습니다.

```
float3 B= {1,0,0};           // 종법선 벡터
float3 T= {0,1,0};           // 접선 벡터
float3 N= normalize(In.Nor);
T = normalize(cross(N, B));
B = normalize(cross(T, N));
```

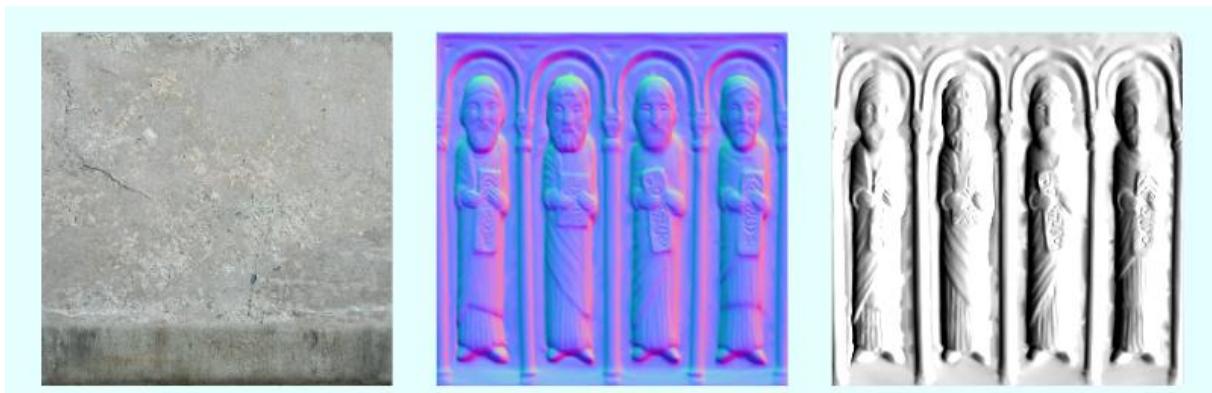
## &lt;ht22\_bump3.zip&gt;

이 방법은 3가지 문제가 발생할 수 있는데 먼저 법선 벡터가 (1,0,0)과 평행이면 평행 벡터의 외적은 0 벡터가 되는 특성에 의해서 접선 벡터가 0 벡터가 됩니다. 0 벡터가 아니라도 두 번째 문제인 접선 벡터의 방향입니다. 무조건 (1, 0, 0) 방향으로 종법선을 설정했기 때문에 반대 방향이 되면 반사의 밝기는 (+)에서 (-)가 되거나 (-)가 (+)가 됩니다. 세 번째 문제는 거의 정사각형 형태에서 법선 맵에서 샘플링 되어야 하는데 밀린 평행사변 형 형태가 되어 텍스처에서 정밀한 법선을 추출하기 어려울 수도 있습니다.

이들 3가지 문제 중에서 2번째가 가장 크게 눈에 띄고 그 다음 첫 번째이며 3번째의 경우는 잘 보이지 않습니다. 이렇게 정점의 법선 만으로 범프 효과를 만드는 것이 문제 점이 있지만 이 효과를 적용 안 하는 것보다 구현하는 것이 훨씬 좋습니다.

[ht22\\_bump4.zip](#)는 정점의 법선 만으로 픽셀 처리과정에서 접선, 종법선을 구해서 범프 효과를 구현한 예입니다. 밝기와 Contrast를 위해서 내적의 결과에 상수를 더했으며 pow() 함수를 이용해서 밝은 부분은 더 밝게, 어두운 부분은 더 어둡게 표현되도록 구현했습니다.

```
Bmp = dot(C1, Lgt);      // 분산 조명 효과
Bmp += 0.85;            // 전체 밝기를 올림
Bmp = pow(Bmp, nPower); // pow() 함수를 사용해서 Contrast를 높임
```



<디퓨즈, 법선 맵, 범프 효과: [ht22\\_bump4.zip](#)>



<디퓨즈 맵 + 범프 효과 - 1.2, 2.2, 4.2 : [ht22\\_bump4.zip](#)>

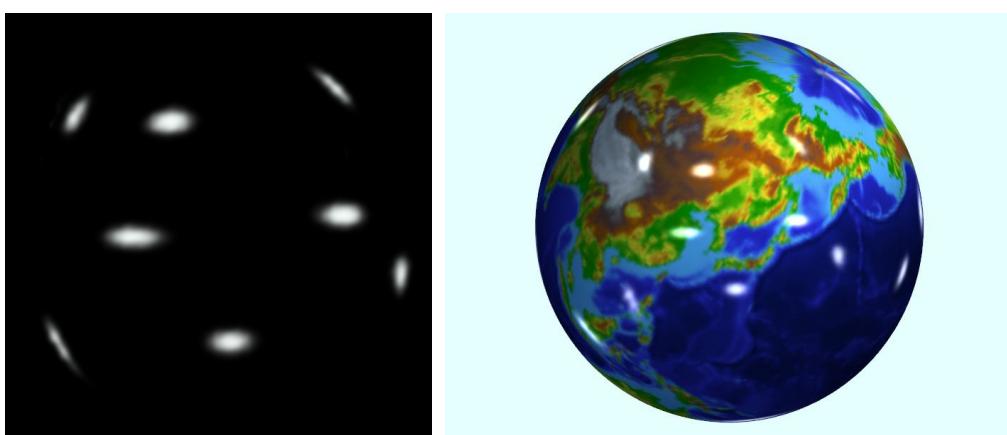
범프 효과를 사용하면 평범한 디퓨즈 맵핑에 입체감 있는 렌더링을 표현할 수 있으며 이것은 이후 Specular 맵핑과 결합되어 좀 더 사실감 있는 효과를 만들어냅니다.

## 5.8 Specular Mapping

스페큘러 맵(Specular Map)은 일종의 Highlight Map으로 스페큘러에 적용되는 반사의 세기와 Sharpness 값이 저장된 텍스처입니다. 풍 쇼이딩 등의 스페큘러 효과를 구현하는 스페큘러 맵핑(Specular Mapping)은 이 텍스처의 값을 가지고 조명의 반사 효과의 변수로 사용합니다.

스페큘러 맵을 사용해서 다음과 같은 간단한 공식으로 최종 색상을 결정할 수도 있습니다.

$$\text{최종 색상} = \text{조명 효과} + \text{Specular Map}$$



<스페큘러 맵과 반사 효과: [ht23\\_spc\\_ati.zip](#)>

조명 효과에 Highlight 색상을 더하는 방법은 구현하기가 쉬어 고정 기능 파이프라인에서부터 자

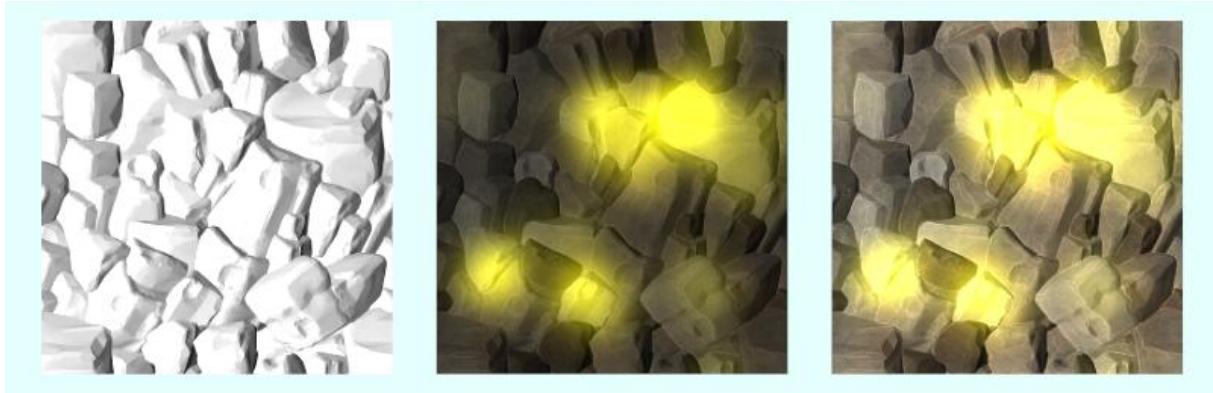
주 사용되던 방법입니다. 처리 방법이 간단해서 대부분의 코드는 조명 효과에 집중되어 있으며 만약 범프 효과와 같이 구현 된다면 최종 색상은 이 둘의 효과를 더해서 만들어 집니다.

최종 색상 = 범프 효과 + Specular Mapping 효과

이 공식에 대해서 고정 기능 파이프라인의 다중 텍스쳐 처리 상태 값은 범프 효과를 먼저 처리하고 그 다음 단계에서 스페큘러 맵의 색상을 더하는 색상 혼합(Color Op)을 ADD로 처리합니다.

```
TEXTUREFACTOR= (m_dTFactor);  
...  
// Tfactor와 법선 맵의 dot 연산  
COLORARG1[0] = TFACTOR;  
COLORARG2[0] = TEXTURE;  
COLOROP [0] = DOTPRODUCT3;  
RESULTARG[0] = CURRENT;  
  
// 디퓨즈 맵과 Modulate 2x 연산  
COLORARG1[1] = CURRENT;  
COLORARG2[1] = TEXTURE;  
COLOROP [1] = MODULATE2X;  
  
// 스페큘러 맵과 Add 연산  
COLORARG1[2] = CURRENT;  
COLORARG2[2] = TEXTURE;  
COLOROP [2] = ADD;
```





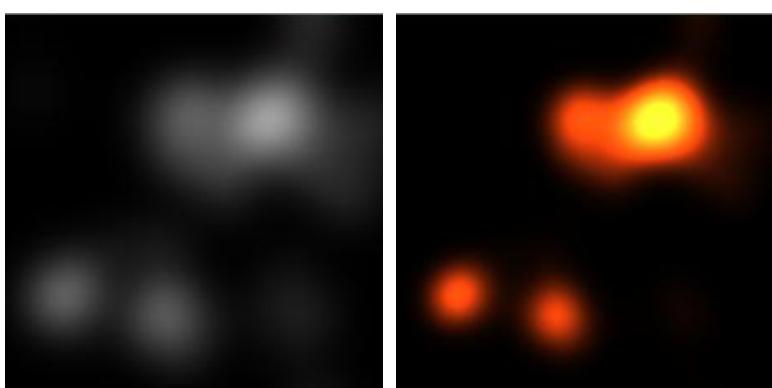
<[ht23\\_spc\\_fixed.zip](#)>

고정 기능 파이프라인에서도 간단하게 구현이 되지만 Highlight에 대한 색상을 정하려면 여러 단계의 멀티 텍스처 처리를 하거나 아니면 간단하게 색상이 있는 스페큘러 맵을 만들어야 합니다. 그런데 쉐이더를 사용하면 단색의 스페큘러 텍스처에 색상을 넣거나 밝기와 Contrast를 높일 수도 있습니다.

```
float4 Px1Prc2(SVsOut In) : COLOR
{
    float4 Hgt = tex2D( SampSpc, In.Tex ); // 스페큘러 맵에서 색상 추출
    float4 Hue={1.0, 0.6, 0.3, 1.0};

    Hgt *= Hue; // 색상 Shift
    Hgt *= 2.5; // 전체 밝기를 높임
    Hgt = pow(Hgt, 2.5)*1.5; // pow() 함수로 Contrast 올림

    return Hgt;
}
```



<쉐이더를 사용해서 색 변환이 가해진 스페큘러 텍스처>

이렇게 단색의 색상에 특정 색상을 곱하고 pow() 함수 등을 사용해서 만든 스페큘러 매핑의 효과를 이전의 범프 효과와 결합하면 좀 더 멋진 효과를 만들어 낼 수 있습니다. 주의 해야 할 것은 곱셈과 pow() 함수의 사용이 많아지면 전체 밝기가 어두워질 수 있습니다. 따라서 픽셀 처리 중간 마다 전체 밝기를 올리는 것이 중요합니다.

스페큘러 매핑과 범프 매핑을 혼합하기 위해서 범프 효과를 만드는 함수를 따로 작성하는 것이 좋습니다.

```
// 범프 효과 처리 함수
float4 NorPrc(float3 In_Nor, float2 In_Tx)
...
N = normalize(In_Nor);
T = normalize(cross(N, B));
B = normalize(cross(T, N));
float3 C = 2*tex2D(SampNor, In_Tx).xyz-1;
...
// 색상에서 추출한 법선 벡터의 회전
C1.x = B.x * C.x + T.x * C.y + N.x * C.z;
C1.y = B.y * C.x + T.y * C.y + N.y * C.z;
C1.z = B.z * C.x + T.z * C.y + N.z * C.z;
...
Bmp = dot(C1, Lgt) + 0.40;
return Bmp;
```

범프 효과를 처리하는 함수와 스페큘러 텍스처의 색상을 처리하는 함수를 만들면 범프 효과 + 스페큘러 매핑 효과를 만들 수 있습니다.

```
float4 PxlPrc3(SVsOut In, uniform float4 Hue) : COLOR
...
float4 TxD = tex2D( SampDif, In.Tex ); // 디퓨즈 텍스처 색상
float4 Bmp = NorPrc(In.Nor, In.Tex); // 범프 밝기
float4 Hgt = tex2D( SampSpc, In.Tex ); // 스페큘러 맵 색상

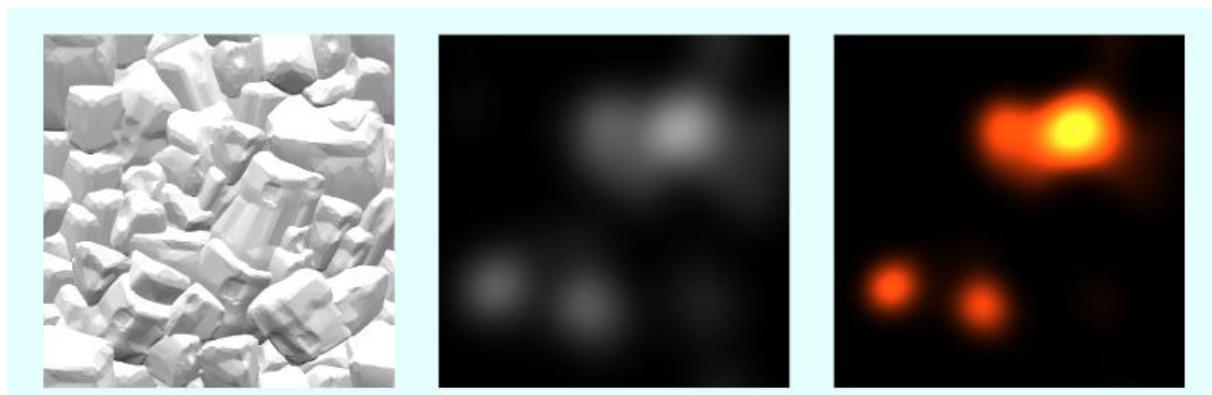
Hgt *= Hue; // 색상 설정
Hgt *= 2.5f; // 밝기 올림
Hgt = pow(Hgt, 2.5)*1.5; // pow() 함수로 Contrast 올림
```

```

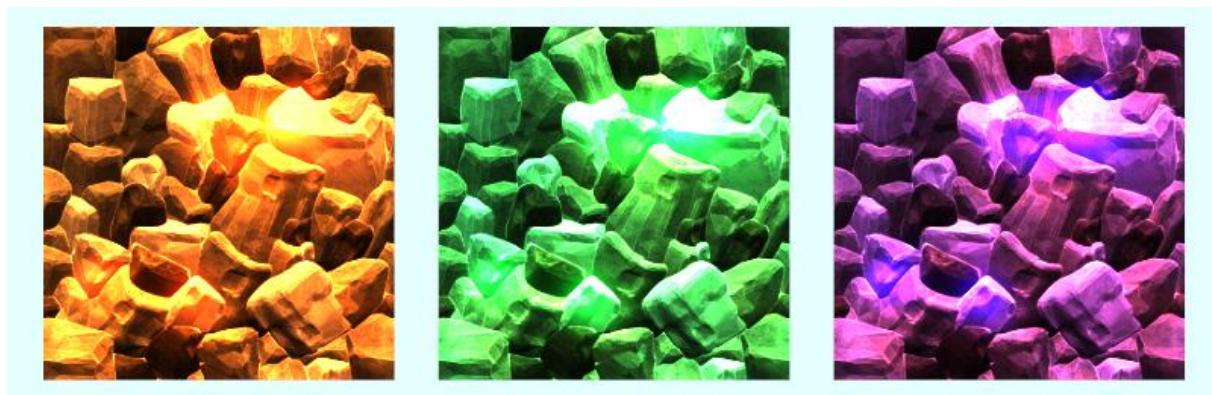
Out = 4*TxD * Bmp;           // 전체 밝기 올림
Out = pow(Out, 2.5);
Out += Hgt;                  // 범프 * 디퓨즈 텍스처 + 스페큘러
Out *=Hue;                  // 색상을 맞춤

return Out;

```



&lt;범프 효과, 스페큘러 효과&gt;

<색상을 가한 범프 효과 + 스페큘러 효과: [ht23\\_spc\\_shader.zip](#)>

실제 렌더링 물체는 평면이 아닌 3차원 입체로 구성되어 있고 정점의 법선 벡터와 빛의 방향 벡터를 이용한 조명의 Specular 효과를 스페큘러 매핑에서 적용해야 합니다. 간단한 방법은 이 둘을 곱해서 Highlight를 만드는 것입니다.

```

float4 PxlPrc(SVsOut In) : COLOR
...
float4 Hgt = tex2D( SampSpc, In.Tex ); // 스페큘러 맵에서 색상 추출
...

```

```

// 조명의 스페큘러 효과 계산
float3 R = normalize(In.Rfc);
float3 E = normalize(In.Eye);
float4 S = saturate(dot(R, E));           // 풍 반사
S = pow(S, m_fShrp);

// 스페큘러 맵 색상과 조명의 스페큘러 반사 세기를 곱함
Hgt *= S;
...
Out += Hgt;                                // 최종 색상에 더함
...
return Out;

```



<Specular 조명 효과를 반영한 범프 + 스페큘러 맵핑: [ht23\\_spc+bump.zip](#)>

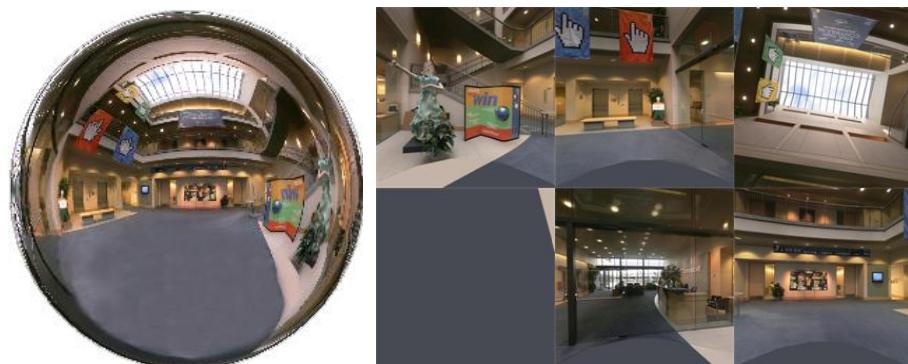
[ht23\\_spc+bump.zip](#) 예제는 조명의 반사 세기를 먼저 계산했지만 스페큘러 맵과 풍 반사 과정의 dot() 처리 결과를 먼저 곱하고 pow() 함수를 적용할 수도 있습니다. 이 방법이 논리적으로 맞아

보이는데 문제는 스페큘러 맵의 반사가 아주 작은 일부의 영역이 되면 Highlight는 거의 보이지 않을 수도 있습니다.

## 5.9 Environment Mapping

환경 매핑(Environment Mapping)은 저 수준 쉐이더와 서피스 강의에서 이미 구현을 해보았습니다. 간단하게 정리한다면 환경 매핑은 3D 물체에 주변의 경관에 대한 반사 또는 굴절을 표현하는 것으로 이를 구현하는 과정은 3D 장면을 텍스처에 저장하고, 다음으로 고정 기능 파이프라인에서 환경 매핑을 구현할 수 있는 상태 값을 설정하거나 아니면 쉐이더로 매핑 텍스처 좌표를 설정합니다.

실시간 장면을 저장하기 위해서 Sphere Map 또는 Cube Map을 사용합니다. Sphere Map은 한 장의 텍스처에 장면을 저장한 것이고 Cube Map은 카메라의 앞쪽, 뒤쪽, 왼쪽, 오른쪽, 위, 아래 6방향에 대해서 장면을 저장한 텍스처입니다.



<Sphere Map과 Cube Map>

단순히 반사에 대해서만 환경 매핑을 구현한다면 Sphere Map이 유리할 수도 있지만 환경 매핑은 반사(Reflection)뿐만 아니라 굴절(Refraction) 효과를 표현하기도 해서 Cube Map을 사용하는 것이 바람직합니다.

아직까지 환경 매핑이 익숙하지 않기 때문에 먼저 저 수준 쉐이더 강의에서 구현한 환경 매핑을 HLSL로 다시 구현해 보겠습니다. 다음으로 Cube Map을 만들어서 반사와 굴절을 표현해 보고 마지막으로 반사, 굴절, 그리고 Diffuse Map을 동시에 적용해서 반 투명 반사 물체를 구현해 보겠습니다.

Sphere Map을 가지고 반사 효과를 만드는 방법은 다음 그림의 붉은 색 화살표에 대한 텍스처 좌표를 정점의 법선 벡터로 사용하면 간단하게 반사 효과를 만들 수 있음을 저 수준 쉐이더 시간에 살펴보았습니다.



```
텍스처 좌표' = 회전 변환된 정점의 법선 벡터 * 뷰 행렬
텍스처 좌표.x = 텍스처 좌표'.x * 0.5 + 0.5
텍스처 좌표.y = -텍스처 좌표'.y * 0.5 + 0.5
```

3D 렌더링 물체는 회전할 수도 있기 때문에 법선 벡터는 먼저 회전 변환을 적용하고 텍스처 좌표로 사용하기 위해서 뷰 변환을 합니다.

```
// 정점 처리 함수
SVsOut VtxPrc(float4 Pos : POSITION0, float3 Nor : NORMAL0)
{
    SVsOut Out = (SVsOut)0; // 출력 데이터 초기화
    ...
    float3 N = Nor;
    float2 T = 1.0;

    N = mul(N, m_mtRot); // 법선 벡터의 회전 변환
    N = mul(N, m_mtView); // 법선 벡터의 뷰 변환

    T = N.xy; // 텍스처 좌표 설정
    T.y = -T.y;
    T = T * 0.5 + 0.5;
```



<Sphere Map 반사: [ht25\\_envl\\_sphere1.zip](#)>

장면을 텍스처에 저장하는 방법은 후면 버퍼의 색상 버퍼(또는 서피스)를 텍스처의 서피스로 대체하는 방법과 ID3DXRenderToEnvMap 객체를 사용하는 방법 두 가지가 있습니다. 이 둘의 사용은 고정 기능 파이프라인의 서피스 강의에 자세히 나와 있으므로 생소한 분들은 그 것을 참고 하기 바라며 여기서는 ID3DXRenderToEnvMap를 사용해서 Sphere, Cube Map에 적용하도록 하겠습니다.

ID3DXRenderToEnvMap는 사용하고 있는 후면 버퍼의 색상, 깊이, 스텐실에 대한 Format을 가지고 D3DXCreateRenderToEnvMap() 함수를 사용해서 객체를 생성합니다. 후면 버퍼의 색상 버퍼는 디바이스의 GetRenderTarget() 함수를 사용해도 되지만 이 함수는 D3DCREATE\_PUREDEVICE로 만들었을 경우에 제대로 동작하지 않을 수 있으므로 GetBackBuffer() 함수를 사용합니다. 깊이-스텐실 버퍼는 GetDepthStencilSurface() 함수를 사용합니다.

```

const int ENVMAP_RESOLUTION = 256;
...

typedef LPD3DXRenderToEnvMap PDRE;
PDRE m_pRndEnv;

HRESULT CMain::Restore()
...
    LPDIRECT3DSURFACE9 pSrf; // 색상, 깊이와 스텐실 버퍼용 서피스
    D3DSURFACE_DESC dscC; // 색상 버퍼 정보
    D3DSURFACE_DESC dscD; // 깊이 버퍼 정보
...
// 색상 버퍼 가져오기
if(FAILED(pDev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &pSrf)))
    return -1;

// 색상 버퍼의 정보 가져오기
pSrf->GetDesc(&dscC);
pSrf->Release();

// 깊이, 스텐실 버퍼 가져오기
if(FAILED(pDev->GetDepthStencilSurface(&pSrf)))
    return -1;

// 깊이, 스텐실 버퍼의 정보 가져오기
pSrf->GetDesc(&dscD);

```

```

pSrf->Release();

// 렌더링 환경 매핑 객체 생성
hr = D3DXCreateRenderToEnvMap( pDev, ENVMAP_RESOLUTION, 1
    , dscC.Format, TRUE, dscD.Format, &m_pRndEnv );

```

장면을 저장하기 위한 Sphere Map은 D3DXCreateTexture() 함수로 생성을 하며 좀 더 빠르게 처리 할 수 있도록 이 함수에 Memory Pool은 D3DPOOL\_DEFAULT로 하고 Usage는 D3DUSAGE\_RENDERTARGET 으로 설정합니다. D3DUSAGE\_RENDERTARGET 옵션이 실패하면 Usage를 0으로 설정하고 다시 생성합니다.

```

typedef LPDIRECT3DTEXTURE9      PDTX;
PDTX    m_pTexSph;
...
// Sphere Map 생성
hr = D3DXCreateTexture( pDev
    , ENVMAP_RESOLUTION, ENVMAP_RESOLUTION
    , 1, D3DUSAGE_RENDERTARGET
    , dscC.Format, D3DPOOL_DEFAULT, &m_pTexSph);

if( FAILED( hr ) )
    hr = D3DXCreateTexture(pDev
        , ENVMAP_RESOLUTION, ENVMAP_RESOLUTION
        , 1, 0
        , dscC.Format, D3DPOOL_DEFAULT, &m_pTexSph );

```

전체 코드는 [ht25\\_env1\\_sphere2.zip](#)의 CMain 클래스를 참고 하기 바랍니다.

이렇게 렌더링 환경 매핑 객체, 그리고 장면을 저장할 텍스처를 만들었으면 3D를 텍스처에 저장하고 환경 매핑 처리에 연결하는 단계만 남아 있습니다.

3D 장면을 텍스처에 저장하기 위해서 카메라의 앞과 뒤에 해당하는 +z, -z, 위, 아래에 해당하는 +y, -y, 그리고 왼쪽, 오른쪽에 해당하는 -x, +x 에 대한 총 6 방향에 대해서 뷰 행렬을 만들고 렌더링을 해야 합니다. D3D SDK의 HDRCubeMap Sample에는 이들 6방향에 대한 뷰 행렬을 만드는 예가 D3DUtility\_GetCubeMapViewMatrix() 또는 DXUTGetCubeMapViewMatrix() 함수로 구현되어 있으며 이들 함수 행렬을 반환하는데 메모리 복사를 조금이라도 피하기 위해 다음과 같이 수정 했습니다.

```
void T_SetupCubeViewMatrix(D3DXMATRIX* pmtView, DWORD dwFace )
```

```

{
    D3DXVECTOR3 vcEye    = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    D3DXVECTOR3 vcLook;
    D3DXVECTOR3 vcUp;

    switch( dwFace )
    {
        case D3DCUBEMAP_FACE_POSITIVE_X:
            vcLook = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
            vcUp   = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
            break;

        case D3DCUBEMAP_FACE_NEGATIVE_X:
            vcLook = D3DXVECTOR3(-1.0f, 0.0f, 0.0f );
            vcUp   = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
            break;

        case D3DCUBEMAP_FACE_POSITIVE_Y:
            vcLook = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
            vcUp   = D3DXVECTOR3( 0.0f, 0.0f,-1.0f );
            break;

        case D3DCUBEMAP_FACE_NEGATIVE_Y:
            vcLook = D3DXVECTOR3( 0.0F,-1.0F, 0.0F );
            vcUp   = D3DXVECTOR3( 0.0F, 0.0F, 1.0F );
            break;

        case D3DCUBEMAP_FACE_POSITIVE_Z:
            vcLook = D3DXVECTOR3( 0.0F, 0.0F, 1.0F );
            vcUp   = D3DXVECTOR3( 0.0F, 1.0F, 0.0F );
            break;

        case D3DCUBEMAP_FACE_NEGATIVE_Z:
            vcLook = D3DXVECTOR3( 0.0F, 0.0F,-1.0F );
            vcUp   = D3DXVECTOR3( 0.0F, 1.0F, 0.0F );
            break;
    }

    // 카메라의 +x, -x, +y, -y, +z, -z 방향에 대한 행렬
    D3DXMatrixLookAtLH(pmtViw, &vcEye, &vcLook, &vcUp );
}

```

이렇게 카메라의 x, y, z 축 방향에 대한 6개의 행렬을 만들고 이를 카메라의 뷰 행렬과 곱하면 각 방향에 대한 뷰 행렬을 만들 수 있습니다.

```
D3DXMATRIX mtViewCur; // 현재 장면의 뷰 행렬
pDev->GetTransform(D3DTS_VIEW, &mtViewCur); // 디바이스에서 뷰 행렬 얻기

D3DXMATRIX mtView[6]; // 카메라의 6 방향에 대한 행렬
for(i=0; i<6; ++i)
{
    T_SetupCubeViewMatrix(&mtView[i], (D3DCUBEMAP_FACES) i );
    mtView[i] = mtViewCur * mtView[i];
}
```

카메라의 각각의 축에 대한 뷰 행렬을 만들었으면 다음으로 6번을 순회하면서 Sphere Map에 장면을 ID3DXRenderToEnvMap 객체의 도움을 받아 렌더링 합니다.

```
// Rendering to Sphere surface
m_pRndEnv->BeginSphere(m_pTxSph);
for(i=0; i<6; ++i)
{
    m_pRndEnv->Face( (D3DCUBEMAP_FACES) i , 0 );
    RenderScene( &mtView[i], &mtPrj );
}
m_pRndEnv->End(0);
```

실시간 장면을 저장한 텍스처를 환경 매핑을 구현한 CShaderEx 객체의 환경 매핑 텍스처로 설정하면 주전자와 표면이 주변을 반사한 효과를 만들 수 있습니다.



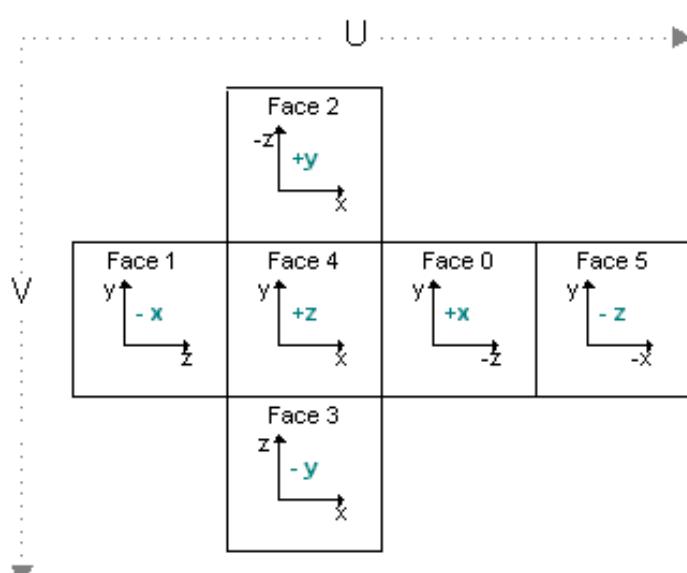
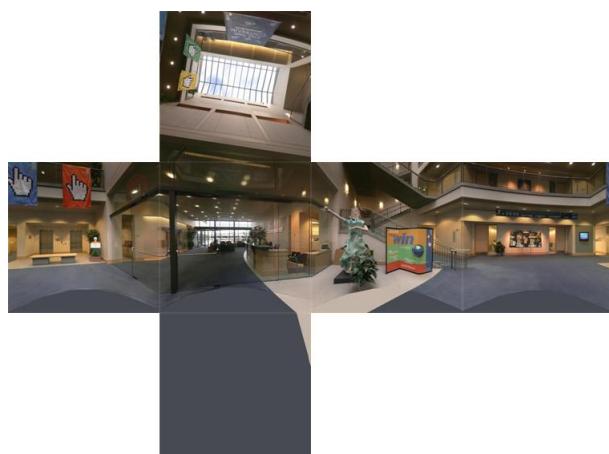
<실시간 Sphere Map: [ht25\\_env1\\_sphere3.zip](#)>

반사 효과는 Sphere Map을 사용해도 충분합니다. 그런데 실 세계의 플라스틱 또는 유리병 같은 물체는 반사(Reflection)뿐만 아니라 굴절(Refraction) 효과도 만들어냅니다.

이것을 3D 구현하려면 Sphere Map으로 더 이상 표현할 수 없습니다. 이렇게 굴절 효과까지 포함한 환경 맵핑은 Cube Map을 사용해야 합니다.

Cube Map은 Sphere Map이 카메라 주변에 대해서 한 개의 텍스처를 사용하는 것에 반해서 각각의 6 방향에 대한 장면을 저장한 텍스처입니다.

반사에 효과 자체는 Cube Map과 Sphere Map 둘 다 비슷하지만 구현하는 방법은 전혀 다릅니다. Cube Map은 Sphere Map과 다르게 그림처럼 카메라의 방향에 따라 장면 그 자체를 저장하고 있어서 적절한 UV를 만들고 텍스처에서 픽셀을 가져와야 합니다.



<Cube Map에 저장된 장면>

UV를 구성하고 픽셀을 가져오는 것이 어려워 보이지만 사용자 프로그래머는 Cube Map을 만들어 놓고 디바이스의 상태 값을 설정하거나 아니면 HLSL의 texCUBE() 함수로 간단히 픽셀을 가져올 수 있어서 이 부분은 걱정할 필요가 없습니다.

Sphere Map을 연습한 것처럼 먼저 미리 만들어진 Cube Map을 HLSL을 사용해서 렌더링 해보고 다음으로 실시간 장면을 Cube Map에 저장해서 렌더링 해봅시다. DX SDK의 예제에는 이미 장면을 저장해 놓은 "LobbyCube.dds"라는 Cube Map 파일이 있습니다.

파일에 저장된 Cube Map의 텍스처는 IDirect3dCubeTexture9 객체가 필요하고 이 객체는 D3DXCreateCubeTextureFromFile() 함수를 사용해서 만듭니다.

```
typedef LPDIRECT3DCUBETEXTURE9 PDTc;
PDTc m_pTxCbm;
D3DXCreateCubeTextureFromFile( m_pDev, "data/LobbyCube.dds", &m_pTxCbm );
```

Sphere Map이 법선 벡터를 직접 텍스처 좌표를 사용했는데 Cube Mapping은 3차원 반사 (Reflection) 벡터를 사용합니다.

이 반사 벡터를 HLSL의 texCUBE() 함수에 전달하면 디바이스는 Cube Map에서 픽셀을 가져옵니다.

```
// Cube 텍스처에서 픽셀을 처리하는 함수
float4 PxlPrc(float3 vcReflection) : COLOR0
{
    return texCUBE( Sampler_CubeMap, vcReflection );
}
```

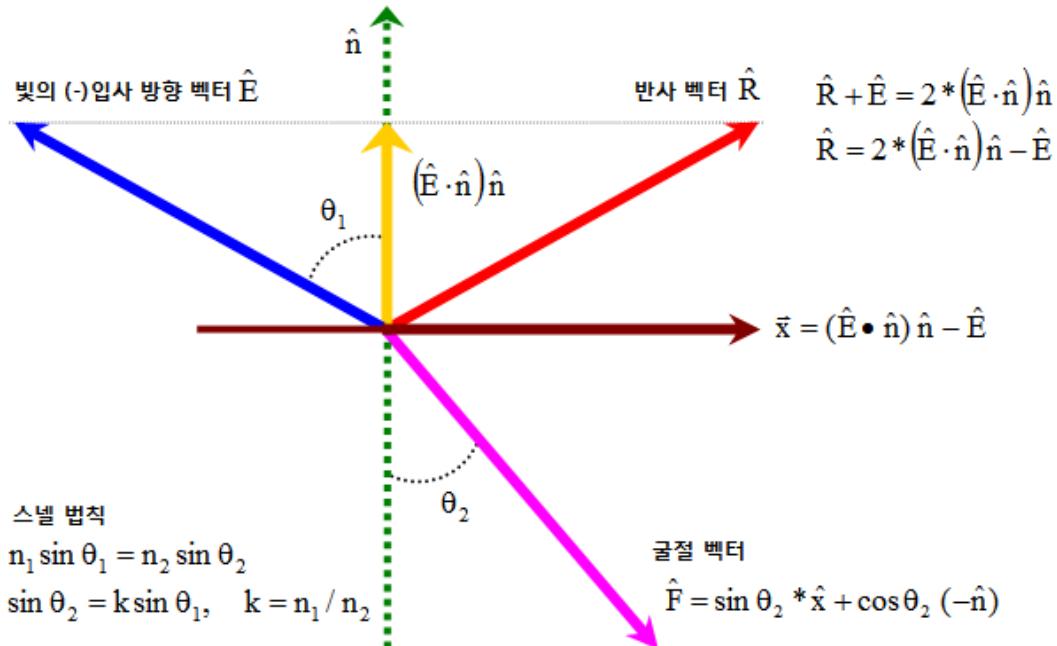
정점 처리와 병행하는 것이 보통이므로 정점 처리에서 변환된 위치와 반사 벡터를 저장할 수 있도록 Cube Mapping에 대한 정점 처리 출력 구조체를 구성합니다.

```
struct SVsOut
{
    float4 Pos : POSITION0;           // 출력 위치
    float3 Rfc : TEXCOORD7;          // 반사 벡터
};
```

이제 정점 처리 함수를 작성해야 하는데 주의할 것은 출력에 대한 반사 벡터는 조명의 반사 중에서 풍 반사 시간에 구현했던 반사 벡터와 거의 같은 방식으로 정점의 법선 벡터와 변환한 정점의 위치에서 카메라의 위치에 대한 방향인 시선 벡터를 사용해서 만듭니다.

그런데 조명과 다르게 이 반사 벡터는 뷰 공간(View Space or Camera Space)의 벡터 이여만 합니

다. 이것은 실시간 장면을 저장한 텍스처는 카메라를 중심으로 만들어지기 때문입니다.



<반사와 굴절 벡터>

뷰 공간에서 반사 벡터를 만드는 방법은 간단하게 정점의 법선 벡터를 회전 변환 후에 다시 뷰 변환 과정을 추가하면 됩니다. 또한 뷰 변환 후 정점의 위치에서 카메라의 위치를 바라보는 시선 벡터는 정점 위치의 월드 변환 후 뷰 변환 후의 벡터를 정규화 하면 됩니다.

```
SVsOut VtxPrc(float3 Pos : POSITION0, float3 Nor : NORMAL0)
{
    SVsOut Out = (SVsOut)0;           // Initialize to Zero
    float4 P = float4(Pos, 1);
    float3 N = Nor;
    float3 E = 1;
    float3 R = 1;

    N = mul(N, m_mtWld);           // 법선 벡터의 월드 변환
    N = mul(N, m_mtView);          // 법선 벡터의 뷰 변환
    N = normalize(N);              // 단위 벡터로 만들

    P = mul(P, m_mtWld);           // 정점 위치의 월드 변환
    P = mul(P, m_mtView);          // 정점 위치의 뷰 변환
```

```

E = -normalize(P);           // 시선 벡터
R = 2.0 * dot(E, N) * N - E; // 반사 벡터: reflect( -E, N );
...
Out.Rfc = R;                // 반사 벡터를 출력 레지스터에 복사

```

HLSL에서 벡터가 3차원 이면 float4x4 행렬과 곱셈을 해도 float3x3 행렬과의 곱셈과 같습니다. 또한 렌더링 물체에 같은 Scale이 적용이 되면 법선 벡터를 월드 변환과 뷰 변환을 진행한 후에 정규화 하면 뷰 공간에서의 법선 벡터가 됩니다.



<장면을 저장한 Cube Map으로 구현한 환경 매펑: [ht25\\_env2\\_cubemap.zip](#)>

Cube Mapping에 대한 HLSL을 작성했고, 다음으로 장면을 Cube Map에 저장하는 단계인데 이 과정은 Sphere Map에서와 거의 같으며 먼저 D3DXCreateCubeTexture() 함수를 사용해서 Cube Map 객체를 생성합니다.

```

hr = D3DXCreateCubeTexture( pDev, ENVMAP_RESOLUTION
                            , 1, D3DUSAGE_RENDERTARGET
                            , dscC.Format, D3DPOOL_DEFAULT, &m_pTexCbm);
if( FAILED( hr ) )
    hr = D3DXCreateCubeTexture(pDev, ENVMAP_RESOLUTION
                            , 1, 0, dscC.Format, D3DPOOL_DEFAULT, &m_pTexCbm);

```

<[ht25\\_env2\\_cube2.zip](#)>

Sphere Map과 마찬가지로 Usage를 D3DUSAGE\_RENDERTARGET 으로 해서 하드웨어 가속을 받게 합니다. 이 옵션이 실패하면 Usage를 0으로 하고 다시 생성합니다. 전체 코드는 [ht25\\_env2\\_cube2.zip](#)을 참고 하기 바랍니다.

이제 남은 단계는 장면을 Cube Map에 저장하는 과정입니다. 이것은 Sphere Map에서와 같으며 차이는 ID3DXRenderToEnvMap 객체의 BeginSphere 대신 BeginCube() 함수를 호출을 시작으로 카메라의 6방향에 따라 장면을 Cube Map에 저장합니다.

```
m_pRndEnv->BeginCube(m_pTxChm);
for(i=0; i<6; ++i)
{
    m_pRndEnv->Face( (D3DCUBEMAP_FACES) i, 0 );
    RenderScene( &mView[i], &mProj );
}
m_pRndEnv->End(0);
```



<Cube Map을 사용한 반사 효과: [ht25\\_env2\\_cube3.zip](#) Key-"1">

Cube Map을 사용하면 굴절 효과도 구현할 수 있습니다. 간단한 파동의 굴절 법칙은 Snell의 법칙을 사용합니다. Snell의 법칙은 매질  $n_1$ 과  $n_2$ 에 대해서 입사각  $\theta_1$ 과 굴절 각  $\theta_2$ 에 대해서 다음과

같은 관계식을 표현한 것입니다.

Snell의 법칙:  $n_1 * \sin \theta_1 = n_2 * \sin \theta_2$

이 법칙을 이용해서 우리는 입사된 빛의 방향 대신 시선 벡터, 법선 벡터, 그리고 매질  $n_1$ 과  $n_2$ 를 가지고 굴절 벡터  $\hat{F}$ 를 구할 수 있습니다.

$$\hat{F} = \sin \theta_2 * \hat{x} + \cos \theta_2 (-\hat{n})$$

$\sin \theta_2$ 는 Snell의 법칙에서 얻고  $\cos \theta_2$ 는  $\sqrt{1 - \sin^2 \theta_2}$ 으로 구합니다. 또한  $\sin \theta_1$ 은  $\hat{x}$  방향에 대한 길이와 같고  $\hat{x}$  방향의 벡터는 법선 벡터와 시선 벡터로 다음과 같이 구할 수 있고 이 벡터의 길이가  $\sin \theta_1$ 가 됩니다.

$$\hat{x} \text{ 방향의 벡터} = (\hat{E} \bullet \hat{n}) \hat{n} - \hat{E}$$

$$\sin \theta_1 = \text{Length}(\hat{x} \text{ 방향의 벡터}) = \text{Length}((\hat{E} \bullet \hat{n}) \hat{n} - \hat{E})$$

$$\sin \theta_2 = k * \sin \theta_1 \quad (k = n_1 / n_2), \quad \cos \theta_2 = \sqrt{1 - \sin^2 \theta_2}$$

이것을 HLSL로 바꾸는 작업이 필요한데 퍼셀 기반 조명에서처럼 정점 처리 과정은 법선과 시선 벡터만 구하고 굴절 벡터는 퍼셀 처리 과정에서 구하기 위해서 정점 처리 결과를 저장할 구조체에 시선 벡터와 법선 벡터를 저장할 수 있도록 선언합니다.

```
struct SVsOut
{
    float4 Pos : POSITION0;
    float3 Eye : TEXCOORD6;           // 시선 벡터
    float3 Nor : TEXCOORD7;          // 법선 벡터
};
```

정점 처리 함수는 법선 벡터의 뷰 공간 변환, 시선 벡터의 뷰 공간 변환만 수행하고 이것을 출력 구조체에 저장합니다.

```
//정점 처리 함수
```

```

SVsOut VtxPrc(float3 Pos : POSITION0, float3 Nor : NORMAL0)
{
    SVsOut Out = (SVsOut)0;
    float4 P = float4(Pos, 1);
    float3 N = Nor;
    float3 E = 1;

    N = mul(N, m_mtWld); // 법선 벡터의 월드 변환
    N = mul(N, m_mtViv); // 법선 벡터의 뷰 공간 변환
    N = normalize(N); // 정규화
    P = mul(P, m_mtWld); // 위치 벡터의 월드 변환
    P = mul(P, m_mtViv); // 위치 벡터의 뷰 변환
    E = -normalize(P); // 시선 벡터는 뷰 변환된 위치 벡터의 정규화와 같음
    ...

    Out.Eye = E; // 시선 벡터 복사
    Out.Nor = N; // 법선 벡터 복사
}

```

픽셀 처리과정은 Snell의 법칙으로 굴절 벡터를 구하고 이 벡터를 HLSL의 texCUBE() 함수의 인수로 전달하는 과정입니다.

먼저 static을 사용해서 두 매질에 대해서 정의 합니다.

```

static float n1 = 1.00; // n1 매질 굴절률
static float n2 = 1.02; // n2 매질 굴절률

// 픽셀 처리함수
float4 Px1Prc(SVsOut In) : COLOR0
...
    float3 E = normalize(In.Eye); // 입력된 시선 벡터의 정규화
    float3 N = normalize(In.Nor); // 입력된 법선 벡터의 정규화
    float3 F = 0; // 굴절 벡터

    float3 X = dot(E, N) * N - E; // 법선에 수직인 x 방향의 벡터
    float sin_theta1 = length(-X); // sinθ1 를 구함
    float k = n1/n2; // 매질의 비율을 구함
    float sin_theta2 = k * sin_theta1; // 스넬 법칙으로 sinθ2 를 구함
    // cosθ2 를 구함
    float cos_theta2 = sqrt(1.0 - sin_theta2 * sin_theta2);
}

```

```

X = normalize(X);           // X 방향의 벡터를 정규화

// 정규화된 X 방향의 벡터, 법선 벡터, cosθ₂, sinθ₂ 를 가지고
// 굴절 벡터를 구함
F = (-N) * cos_theta2 + X * sin_theta2;

Out = texCUBE( SmpCbm, F);

```

Snell의 법칙을 사용해서 굴절 벡터를 직접 구현 했는데 HLSL은 반사에 대한 reflect() 함수가 있듯이 굴절에 대한 refract() 함수가 있습니다. reflect() 함수와 refract() 함수를 사용해서 이전의 쉐이더 코드를 다음과 같이 대체할 수 있습니다.

```

float k = n1/n2;           // 굴절 비율
R = reflect(-E, N);        // 반사 벡터
F = refract(-E, N, k);     // 굴절 벡터

```

[ht25\\_env2\\_cube3.zip](#) 는 Cube Map을 이용해서 반사와 굴절 효과를 표현한 예입니다. 숫자 키 1을 누르면 반사효과를, 숫자 키 2를 누르면 굴절 효과를 볼 수 있습니다.



<Cube Map을 사용한 굴절 효과: [ht25\\_env2\\_cube3.zip](#) Key-"2">

Cube Map은 반사와 굴절 모두를 표현할 수 있어서 플라스틱 병과 같은 거의 투명한 물체를 쉽게

표현할 수 있습니다. 이런 물체들은 빛의 입사각이 작으면 투과율이 높고 입사각이 크면 반사율이 높습니다.

입사각은 시선 벡터와 법선 벡터의 내적으로 구할 수 있으므로 투과율(또는 비중)을 이 둘의 벡터의 내적의 제곱으로 간단히 결정해 보도록 합시다.

투과율(w) :

```
w = dot(반사 벡터 또는 시선, 법선 벡터);
w = w*w;
```

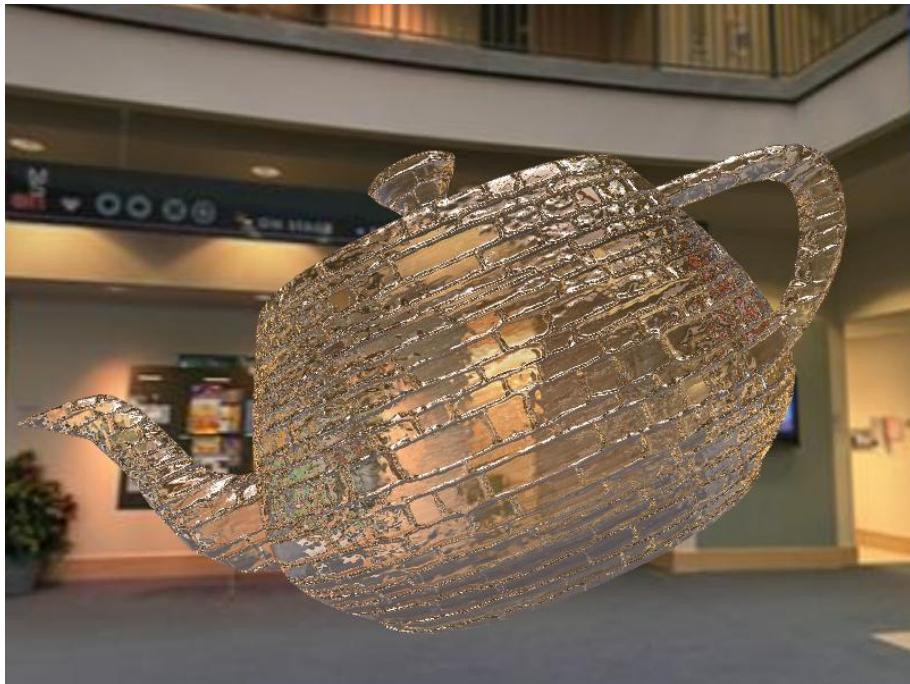
또한 렌더링 물체에 약간의 Diffuse Map을 적용하면 사실감을 더 높일 수 있으며 [ht25\\_env2\\_cube4.zip](#)는 반사+굴절 효과에 Diffuse Map을 20% 적용해서 구현한 예제입니다.



<Cube Map을 사용한 반사 + 굴절 + Diffuse Map 효과: [ht25\\_env2\\_cube4.zip](#)>

Cube Map을 사용한 환경 매핑은 법선 벡터를 사용하기 때문에 이 법선 벡터에 대해서 Normal Map을 적용하면 올록볼록한 표면의 반사와 굴절을 만들 수 있습니다. [ht25\\_env2\\_cube5.zip](#)는 Normal Map을 법선 벡터로 사용해서 환경 매핑을 구현한 예제입니다.

텍스처에서 법선 벡터를 구하는 함수는 이전의 범프 효과(Bump Effect)에서 사용한 함수를 거의 그대로 사용했습니다.



<Cube Map + Normal Map을 사용한 반사 + 굴절 효과: [ht25\\_env2\\_cube5.zip](#)>

## 5.10 Water Reflection Effect

환경 매핑(Environment Mapping)은 저 수준 쇼이더와 서피스 강의에서 이미 구현을 해보았습니다. 범프 맵을 결합한 환경 매핑의 응용으로 물에 대한 반사 효과를 만들 수 있습니다. 물에 대한 효과는 과동 방정식(Wave Equation), 반사, 그리고 굴절에 대한 적절한 수식을 필요하기 때문에 향상된 기술을 선보이기 위해서 3D의 예제로 가장 많이 구현되고 있습니다.

보통 과동 방정식의 미분 방정식 형태는 다음과 같이 주어집니다.

$$\nabla^2 \Psi = \frac{1}{V^2} \frac{\partial^2}{\partial t^2} \Psi$$

이 미분 방정식을 풀기 위해서 조화 진동자(Harmonic Oscillator) 모델을 사용하고 있으며 점성(Damping)이 있는 조화 진동자의 풀이는  $\exp()$  함수의 결합 형태로 풀이가 됩니다.

$$e^{(-kx-iwt)}$$

또한 푸아송 방정식에 의해서 하나의 과동은 여러 과동의 중첩(Super Position)으로 풀이가 가능하고 이 것을 적용하면 과동에 대한 최종 해는 각각의 과동을 더한 결과가 됩니다.

$$\sum_j e^{(\vec{k}_j \cdot \vec{x} - i w_j t)}$$

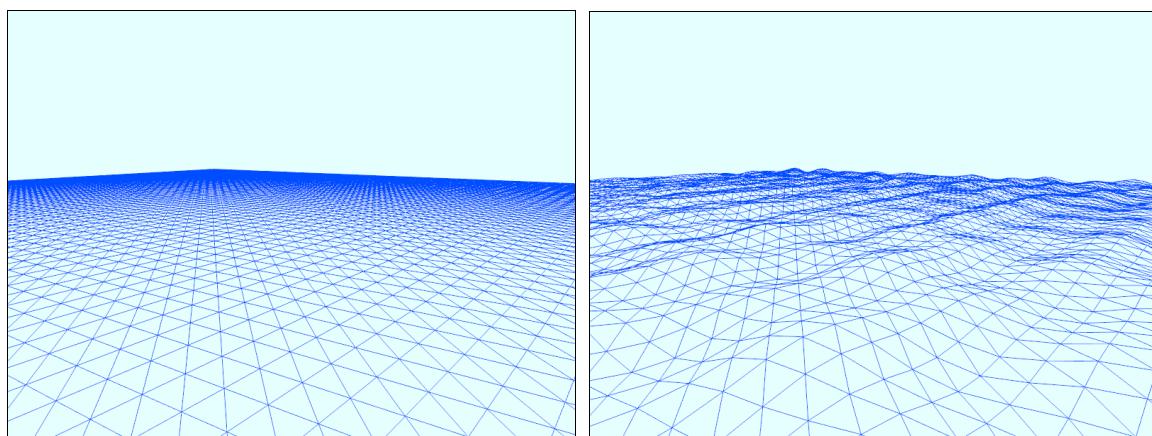
바다와 호수 등의 파동을 만드는 주요 요인은 바람입니다. 이 바람을 이용해서 수면의 운동을 구현하는 것이 가장 바람직할 수 있지만 여기서는 예제 수준 정도로만 만들어 보기 위해서 4개의 돌이 위 아래로 움직이고 이 돌에 의해 파동이 만들어지는 것을 가지고 물에 대한 효과를 만들어 보겠습니다.

물 분자의 운동은 격자가 일정한 정점의 위, 아래 움직임으로 표현할 수 있습니다. 그런데 정점 벼페 또는 시스템 메모리에 만든 정점의 모든 위치를 직접 변경하는 것은 교체에 대한 부담이 큽니다. 따라서 정점 쉐이더에서 출력 위치의 수식을 만들어서 바꾸는 것이 좋습니다.

$e^{(\vec{k}_j \cdot \vec{x} - i w_j t)}$ 의 간단한 형태는  $e^{(-k_j * x)} \sin(w_j t - \phi_j)$  가 되고 이것을 HLSL로 쉽게 작성할 수 있습니다.

```
float3 WavePos(float3 Pos, float3 eps=float3(0,0,0))
...
tPos.y += exp(-wvK.x * r)*sin(r * wvOmega.x - m_fTime * wvSpeed.x);
...
```

이 HLSL을 [ht25\\_env2\\_water1.zip](#) 예제에 적용하면 [ht25\\_env2\\_water2.zip](#)와 같이 물결이 출렁이는 효과를 볼 수 있습니다.



<수면의 파동. [ht25\\_env2\\_water1.zip](#), [ht25\\_env2\\_water2.zip](#)>

정점의 움직임을 완성했고 다음으로 수면의 반사와 굴절을 처리할 차례입니다. 이것은 이전의

범프 매핑을 거의 그대로 이용하는 것이 좋습니다. 동적인 효과를 만들기 위해서 정점의 UV 좌표가 시간에 의존하게 하는 것이 좋으며 또한 한 방향 보다 여러 방향으로 설정하는 것이 더 효과적입니다. 이를 정점 처리 함수에 적용합니다.

```
SVsOut VtxPrc(float3 iPos : POSITION0)
...
float2 wvSpdU=float2(0.02f, +0.02f);
float2 wvSpdV=float2(0.02f, -0.02f);
float Time = m_fTime;
float2 Tex= 1;
float2 Ds1; // Distortion UV1
float2 Ds2; // Distortion UV2

Tex.x = iPos.x/16.;
Tex.y = 1- iPos.z/16.;
Ds1 = Tex.xy + wvSpdU * Time; // 시간에 의존하는 텍스처 좌표(UV1) 생성
Ds2 = Tex.yx + wvSpdV * Time; // ...
Out.Ds1 = Ds1;
Out.Ds2 = Ds2;
...
```

빛의 반사와 굴절은 프레넬(Fresnel) 방정식으로 풀이 되며 프레넬 방정식의 근사식은 다음과 구할 수 있습니다.

Fresnel 근사식 =  $F + (1-F)*\cos\theta^5$

게임은 현실의 적당한 흉내내기 이므로 굴절률 n1, n2가 주어질 때 좀 더 간단한 형태의 프레넬 방정식을 만들 수 있습니다.

Simple Fresnel  $F = \text{pow}(\frac{n2}{n1} \cdot \text{dot}(-E, N))$

이 근사식을 픽셀 처리 함수에 적용해서 반사 계수로 사용해서 굴절 효과까지 만들어야 하는데 여기서는 반투명 계수 정도로만 사용하도록 하겠습니다.

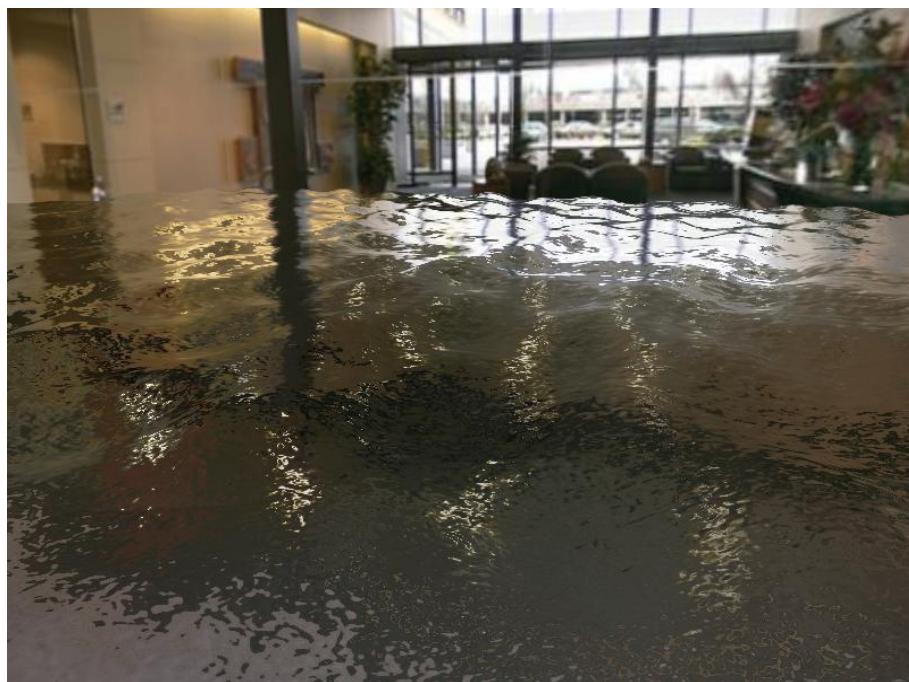
```
float4 PxlPrc(SVsOut In) : COLOR0
...
float3 E = normalize(In.Eye);
```

```

float F;
float n1      = 1.0;
float n2      = 1.333;
...
// Calculate Simple Fresnel = (F - I.N)^2
F = pow(F - dot(-E, N), 2);
Out = texCUBE( SmpCbm, R);
Out.a = F;
...

```

내용은 약간 길지만 어렵지 않으므로 전체 코드는 [ht25\\_env2\\_water4.zip](#)를 참고하길 바랍니다.



<수면 반사 효과: [ht25\\_env2\\_water4.zip](#)>

## 5.11 깊이 버퍼 그림자

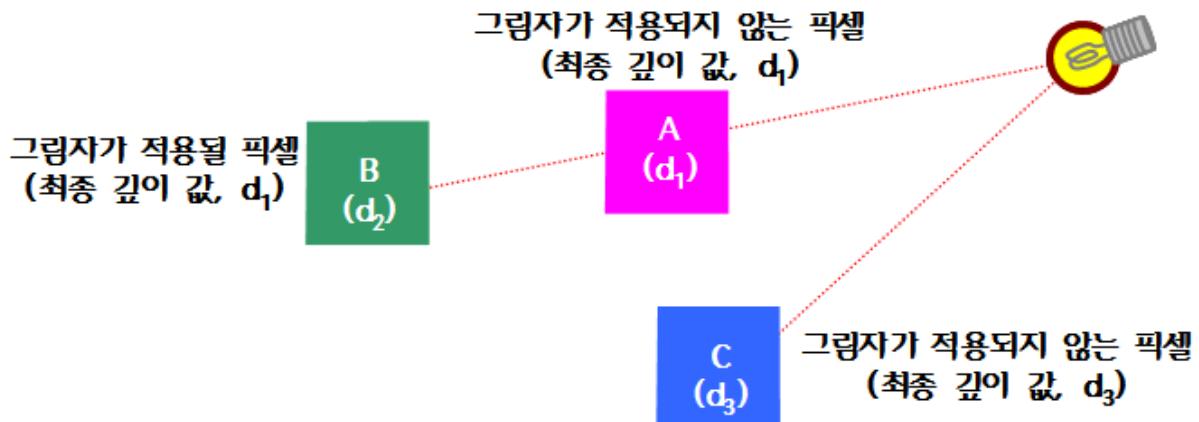
그림자를 게임의 3D 장면에 구현하는 것은 게임을 만드는 과정 중에서 프로그래머에게 기쁨을 주는 작업 중의 하나입니다. 현재 그림자를 만드는 방법은 간단한 원형 이미지를 캐릭터의 뒷 밑에 렌더링하거나 3D 물체를 2차원 평면 텍스처에 저장하고 이 텍스처를 매핑 하는 투영 그림자 매핑 방법이 있습니다. 하드웨어 성능이 좋다면 좀더 향상된 방법으로 광원의 위치와 방향에 대해서 3D

물체의 깊이를 텍스처로 저장하고 이 텍스처의 깊이에 따라 그림자를 표현하는 깊이 버퍼 그림자가 있습니다. 또한 오래 전부터 사용되고 있으며 가장 멋진 그림자를 만드는 부피 그림자(Volume Shadow)가 있습니다. 이 방법은 이전 3D 기초 시간에서 프레임 버퍼의 스텐실 버퍼를 사용해서 구현해 보았습니다.

하드웨어, 3D 장면에 소모되는 그래픽 리소스, 장르 등에 따라서 간단한 그림자를 선택하거나 아니면 사실감 있는 그림자를 만들 수 있는데 그림자를 표현하지 않는 것보다 어떤 식으로든 표현하는 쪽이 게임의 사실감을 더 높여 줍니다. 이것은 조명과 비슷해서 조명을 사용함으로써 부피 느낌을 만들 듯이 그림자는 객체와 주변의 환경에 대해서 공간 느낌을 형성하기 때문입니다.

여러 가지의 그림자를 구현 방법 중에서 원형 그림자는 구현 방법이 쉽기 때문에 그냥 넘어가겠습니다. 대신 쉐이더를 사용해야 쉽게 만들 수 있는 깊이 버퍼 그림자를 먼저 구현해 보도록 하겠습니다.

깊이 버퍼 그림자의 원리는 의외로 간단합니다. 그림처럼 만약 현재의 정점과 조명 사이에 어떤 정점이 존재하면 디퓨즈 색상을 어둡게 처리하는 것입니다. 예를 들어 그림의 A, B, C 픽셀을 조명에서 바라본 깊이 값을  $d_1$ ,  $d_2$ ,  $d_3$ 로 계산이 되었다고 합시다. 다음으로 조명에서 바라본 최종 깊이 값을 결정할 때 픽셀 B는 픽셀 A로 가려지기 때문에 최종 깊이 값을  $d_2$ 가 아닌  $d_1$ 을 가지고 옵니다.



<깊이 버퍼 그림자의 원리>

깊이 버퍼 그림자의 원리는 이렇게 조명에서 바라본 최종 깊이 값을 먼저 구성하고 다시 원래의 깊이 값을 비교를 하는 것입니다. A와 C 픽셀은 깊이 값의 변화가 없지만 픽셀 B는 자신의 깊이 값  $d_2$ 보다 최종 깊이 값  $d_1$ 이 작으므로 그림자 적용 대상이 되는 픽셀이 되는 것입니다.

간단한 내용이지만 어떻게 조명에서 바라본 최종 깊이 값을 만들 수 있을까요? 우리는 이전 쉐이더 기초 시간에 장면의 깊이 값을 텍스처에 저장하는 방법을 알고 있습니다. 즉, 디바이스의 파이프 라인을 이용해서 조명의 위치와 방향에 의존하는 뷰 행렬과 투영 행렬을 가지고 뷰 변환, 정

규 변환에 적용해서 장면에 사용된 물체들의 위치를 텍스처에 저장하는 것입니다.

그 다음으로 조명의 위치로부터 깊이를 다시 계산하고 이 값을 깊이가 저장된 텍스처의 값과 비교해서 값이 텍스처의 값보다 크면 그림자가 적용되는 픽셀로 판정을 합니다. 이 판정을 위해서 정점 쉐이더에서는 조명의 뷰 행렬, 투영 행렬로 렌더링 물체의 깊이를 계산하고 이 값을 픽셀 쉐이더로 넘깁니다. 픽셀 쉐이더 함수는 깊이가 저장된 텍스처에 색상을 추출해서 정점 처리에서 넘어온 값과 비교하는 코드를 작성합니다.

지금까지 깊이 버퍼 그림자를 구현 하는 내용을 간단히 살펴보았습니다. 이 내용을 구체적으로 구현하도록 하겠습니다. 첫 번째 해야 할 일은 조명의 위치와 방향으로 미리 저장된 텍스처가 존재한다는 가정 하에 이것을 그림자가 적용될 물체에 매핑하도록 하는 것입니다. 이 방법은 3D 기초 시간에 연습했던 투영 매핑과 동일 하며 우리는 모든 처리를 쉐이더를 사용할 것이므로 고정 기능 파이프라인에서 구현된 투영 매핑을 HLSL로 변환해 보는 것이 중요합니다.

모델 좌표계에 존재하는 정점을 화면에 연출하기 위해서 우리는 3D 장면을 구성하는 월드의 행렬을 적용한 월드 변환, 카메라의 공간으로 변환하는 뷰 변환, 그리고 정규 또는 투영 변환을 작성해야 합니다. 투영 매핑은 장면의 뷰, 투영 행렬 대신 조명의 위치와 방향으로 구성된 조명의 뷰 행렬과 투영 행렬을 사용하고 이 변환을 거친 위치를 텍스처 좌표로 사용하는 것이며 이 과정을 위해서 다음과 같이 최소한 5개의 행렬이 필요합니다.

```
float4x4 m_mtWld;           // 월드 변환 행렬
float4x4 m_mtViw;          // 카메라 뷰 행렬
float4x4 m_mtPrj;          // 3D 장면의 투영 행렬
float4x4 m_mtSdV;          // 조명의 위치와 방향으로 만든 뷰 행렬
float4x4 m_mtSdP;          // 조명의 투영 변환 행렬
```

속도의 향상을 위해서 월드 변환 행렬 \* 카메라 뷰 행렬 \* 3D 장면의 투영 행렬을 미리 곱한 행렬과 월드 변환 행렬 \* 조명의 뷰 행렬 \* 조명의 투영 행렬을 곱한 2개의 행렬을 쉐이더로 전달할 수도 있지만 여기는 구현의 내용에 초점을 두었기 때문에 5개의 행렬을 사용하고 있습니다.

정점 쉐이더 함수는 정점의 위치를 입력 받고, 이 위치를 가지고 출력 위치와 텍스처 좌표를 만듭니다.

```
void VtxPrc(in float4 iPos : POSITIONO // 입력: 정점의 위치
            , out float4 oPos : POSITIONO // 출력: 변환된 정점의 위치
            , out float2 oTex : TEXCOORD0 // 출력: 텍스처 좌표
...

```

```

PosW = mul(iPos, m_mtWld);
PosT = mul(PosW, m_mtView);           // 3D 장면에 대한 뷰 변환
PosT = mul(PosT, m_mtProj);          // 3D 장면에 대한 투영 변환
oPos = PosT;                         // 출력 위치 설정

```

이 곳까지의 HLSL 코드는 정점의 월드, 뷰, 투영 변환에 해당합니다. 다음으로 텍스처의 매핑에 사용되는 UV 좌표를 설정하는 단계인데 장면 연출과 동일하게 월드 변환을 진행하고 다음으로 조명에 의존하는 뷰와 투영 행렬 변환을 수행합니다.

```

PosT = mul(PosW, m_mtSdV);           // 조명에 대한 뷰 변환
PosT = mul(PosT, m_mtSdP);          // 조명에 대한 투영 변환

```

3D에서 그래픽 파이프라인의 정규 또는 투영 변환을 거치면 x, y 값은 [-1, 1] 범위의 값으로 정규화 됩니다. 그런데 텍스처 좌표는 [0, 1] 이므로 정규 변환을 통과한 x 값은 [0, 1], y 값은 [1, 0] 범위로 조정해야 합니다. 조정된 값을 텍스처 좌표로 출력 레지스터에 쓰기만 하면 정점의 위치를 깊이 버퍼 그림자의 텍스처 좌표로 만드는 과정이 끝나게 됩니다.

```

PosT.x = (1.0 + PosT.x) * 0.5F; // x 위치 [-1,1]에서 [0,1]로 변환
PosT.y = (1.0 - PosT.y) * 0.5F; // y 위치 [-1,1]에서 [1,0]로 변환
oTex = PosT;                  // 텍스처 좌표 설정

```

이 과정도 행렬을 사용할 때도 있습니다. 이 때 여러분은 다음과 같이 행렬을 만들어서 PotT에 곱해야 합니다.

```

D3DXMATRIX mtTex(0.5F, 0.0F, 0.0F, 0.0F,
                  0.0F, -0.5F, 0.0F, 0.0F,
                  0.0F, 0.0F, 1.0F, 0.0F,
                  0.5F, 0.5F, 0.0F, 1.0F );

```

텍스처 좌표가 [0, 1] 넘어서는 값들은 그림자를 적용할 필요가 없음으로 샘플러의 어드레스 모드를 Border 또는 Clamp로 설정할 수 있지만 Border로 설정하는 것이 좋고 Border 색상을 0xFFFFFFFF로 정합니다. 쉐이더의 색상은 1.0이고 정규화된 깊이는 최대 1.0이기 때문에 0xFFFFFFFF 값은 깊이 값을 1로 설정하는 것과 같은 의미입니다.

```

sampler SmpDif : register(s0) = sampler_state
...
AddressU      = Border;

```

```

AddressV      = Border;
BorderColor   = 0xFFFFFFFF;
};


```



<이미지 투영: 정점 위치를 텍스처 좌표계로 사용한 예. [ht26\\_shadow0.zip](#)>

텍스처를 깊이 버퍼 그림자 매핑을 만들어 보았습니다. 이제 깊이 버퍼 그림자의 첫 번째 단계인 깊이 값 저장을 구현해 보겠습니다.

게임에서 깊이는 최소한 16비트 이상을 사용합니다. 따라서 이 정도의 깊이를 저장할 수 있는 해상도가 높은 텍스처를 사용해야 하는데 R8G8B8 형식의 텍스처보다 R16G16B16, R32G32B32 형식의 텍스처를 선택하거나 아니면 단일 색상으로 32비트 정보를 저장할 수 있는 D3DFMT\_D32F 형식의 텍스처를 깊이 텍스처로 선택하는 것이 중요합니다. 그리고 텍스처를 생성하기 위해서 여러분은 D3D Device의 멤버 함수 CreateTexture() 함수를 사용하는 것보다 D3DXCreateTexture() 함수를 사용하는 것이 안전합니다.

```
D3DXCreateTexture(..., 1, D3DUSAGE_RENDERTARGET, D3DFMT_R32F, ...);
```

쉐이더 코드는 정점의 위치를 조명에 대한 뷰 행렬과 투영 행렬을 적용해서 1차원 텍스처 좌표로 출력 합니다.

```

void VtxShadowMap( in float4 iPos : POSITION
                    , out float4 oPos : POSITION
                    , out float oTex : TEXCOORD0
                    ...
                    )
{
    Pos = mul(iPos, m_mtWld);
    Pos = mul(Pos, m_mtSdV);           // 조명에 대한 뷰 변환
    Pos = mul(Pos, m_mtSdP);           // 조명에 대한 투영 변환
}

```

```

oPos = Pos;
oTex = Pos.z/Pos.w;
}

```

이 HLSL 코드는 3D 장면 연출에서 사용되는 변환 과정과 동일하고 단지 차이라면 마지막 줄에서 텍스처 좌표를 변환 된 위치의 "z"값을 "w"값으로 나누는 것입니다. 이렇게 하면 텍스처의 좌표는 [0, 1] 값으로 정규화 됩니다. 때로는 정규화 시키지 않고 그대로 픽셀 쉐이더로 넘기는 것도 생각할 수 있지만 다른 작업과 협업을 생각하면 정규화 하는 것이 좋습니다.

이렇게 정점에서 처리한 1차원 깊이 값을 픽셀 쉐이더 함수는 전달 받아서 이 1차원 좌표 값을 색상으로 그대로 출력합니다.

```

float4 PxShadowMap(float Tex : TEXCOORD0) : COLOR0
{
    return Tex;
}

```

만약 여러분이 Target을 R8G8B8 형식을 사용했다면 이 부분에서 깊이 값들이 유실 될 수 있어서 그림자 처리를 제대로 수행 못할 수 있게 됩니다. 또한 정점 처리 함수에서 변환된 깊이 값의 Semantic을 "COLOR#"으로 설정하지 않고 "TEXCOORD#"를 사용한 것은 특정 그래픽 카드는 "COLOR#"로 Semantic을 설정하면 정점 처리 후에 픽셀 단계로 전달 할 때 [0, 1] 범위로 정규화 하는 것도 있기 때문입니다.

이렇게 조명에서 바라본 정점의 최종 깊이 값을 파이프라인과 쉐이더를 사용해서 만들었습니다. 다음 단계는 최종 깊이 값과 현재의 깊이 값을 비교해서 그림자를 적용할 차례입니다. 픽셀 처리로 투영 매핑 좌표와 깊이 값을 저장할 수 있는 구조체를 선언 합니다.

```

struct VsOut
{
    float4 Pos : POSITION0;
    float4 Dif : TEXCOORD1;           // Diffuse 색상
    float2 Shd : TEXCOORD2;          // 그림자 UV
    float Dpc : TEXCOORD3;           // 조명에서 바라본 현재의 깊이 값
};

```

앞서 현재의 깊이 값과 최종 깊이 값의 비교를 위해서 정점 처리 함수에서 현재의 깊이 값을 계산한다고 했습니다. 현재의 깊이 값은 조명의 뷰와 투영 행렬로 결정을 해야 합니다.

```

VsOut VtxShadowScene(float4 iPos : POSITION0, float3 iNor : NORMAL0)
...
    PosW = mul(iPos, m_mtWld);           // 위치의 월드 변환
...
    PosT = mul(PosW, m_mtSdV);           // 조명에 대한 뷰 변환
    PosT = mul(PosT, m_mtSdP);           // 조명에 대한 투영 변환
    Out.Dpc = (PosT.z-0.01)/PosT.w;      // shift: z-bias

    PosT.x = (1.0 + PosT.x) * 0.5;       // 투영 텍스처 좌표 X
    PosT.y = (1.0 - PosT.y) * 0.5;       // 투영 텍스처 좌표 Y
    Out.Shd = PosT;                      // 투영 텍스처 좌표 출력

```

정점의 투영 변환 후에 "-0.01"를 더한 것은 픽셀 처리에서 텍스처에 저장된 최종 깊이 값과 비교를 할 때 비교 오차로 인해서 줄 무늬가 나타날 수 있기 때문입니다. 이를 위해서 z 값을 적당히 이동 시키는 z-bias가 필요하며 이 값을 대충 "-0.01"로 설정한 것입니다. 정교한 프로그램이라면 이 부분도 깊이 값에 의존하도록 작성해야 됩니다.

픽셀 처리 함수는 정점 투영 텍스처 좌표를 가지고 최종 깊이를 저장한 텍스처에서 깊이 값을 가져옵니다. 다음으로 이 값을 정점 처리에서 전달된 깊이 값과 비교를 합니다. 만약 정점 처리에서 전달된 값이 텍스처에서 추출한 값보다 크면 그림자를 적용할 픽셀로 결정이 됩니다. 다음 쉐이더 코드에서는 이 값을 0.0으로 정했습니다.

```

float4 Px1ShadowScene(VsOut In) : COLOR0
{
    float4 Out = 0;                      // 투영 텍스처 좌표 출력
    float Shd = 0;                       // 그림자 유무

    // 텍스처에서 조명에서 바라본 최종 깊이 값 추출
    Shd = tex2D(SmpShd, In.Shd);

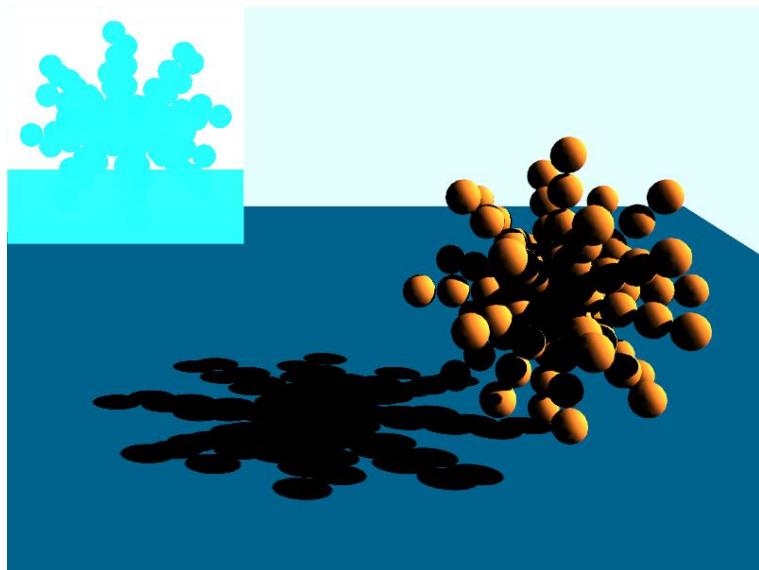
    // 정점 처리에서 만든 깊이 값과 비교
    // 정점 처리의 값보다 작으면 그림자를 적용할 대상으로
    // 색상을 0으로 함
    if(Shd >= In.Dpc)
        Shd = 1.0;
    else

```

```

Shd = 0.0;
Out = Shd * In.Dif;
...

```



<깊이 버퍼 그림자. [ht26\\_shadow1.zip](#)>

그림자가 적용된 부분을 확대하면 Aliasing을 볼 수 있습니다. 좀 더 부드러운 그림자를 만들기 위해서 픽셀 처리에서 9-CON Sampling 등으로 간단하게 해결 할 수도 있습니다. 9-CON 샘플링은 현재의 픽셀과 인접한 8개의 픽셀을 혼합하는 방법입니다. 계산을 빨리 하기 위해서 2차원 좌표로 구성된 9개의 좌표가 필요합니다. 다음의 9-CON 샘플링 테이블은 깊이 버퍼 그림자 텍스처의 사이즈가 1024x1024이기 때문에 인접한 픽셀을 얻기 위해서 1/1024.0 값을 사용하고 있습니다.

```

// 9-CON 샘플링 테이블
static float2 c[9] =
{
    {-1./1024., -1./1024.}, {-1./1024., 0./1024.}, {-1./1024., 1./1024.},
    { 0./1024., -1./1024.}, { 0./1024., 0./1024.}, { 0./1024., 1./1024.},
    { 1./1024., -1./1024.}, { 1./1024., 0./1024.}, { 1./1024., 1./1024.},
};

```

9-CON 샘플링의 적용은 9번 샘플링 해서 이 결과를 가지고 누적시켜서 그림자 적용을 결정합니다.

```

float4 PxlPrc(VsOut In, uniform bool bTex=true) : COLOR
{

```

```
float4 Out = 0;  
float Shd = 0;  
float r = 0;  
  
for(int i=0;i<9; ++i)  
{  
    r = tex2D(SmpShd, In.Shd + c[i]);  
    if(r >= In.Dpc)  
        Shd += 1.0;  
}  
  
Shd *= 0.111f;  
Out = Shd * In.Dif;  
...
```

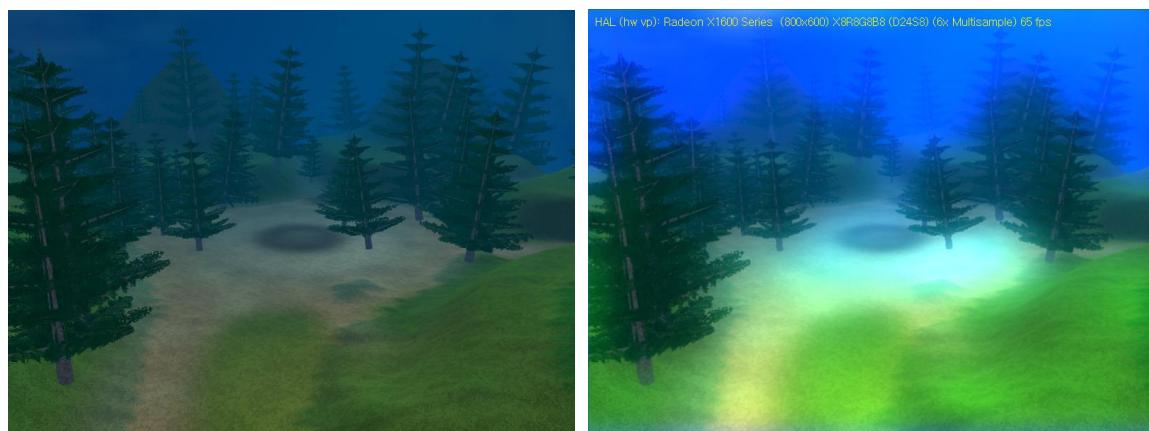


<9-CON 샘플링이 적용된 깊이 버퍼 그림자. [ht26\\_shadow2.zip](#)

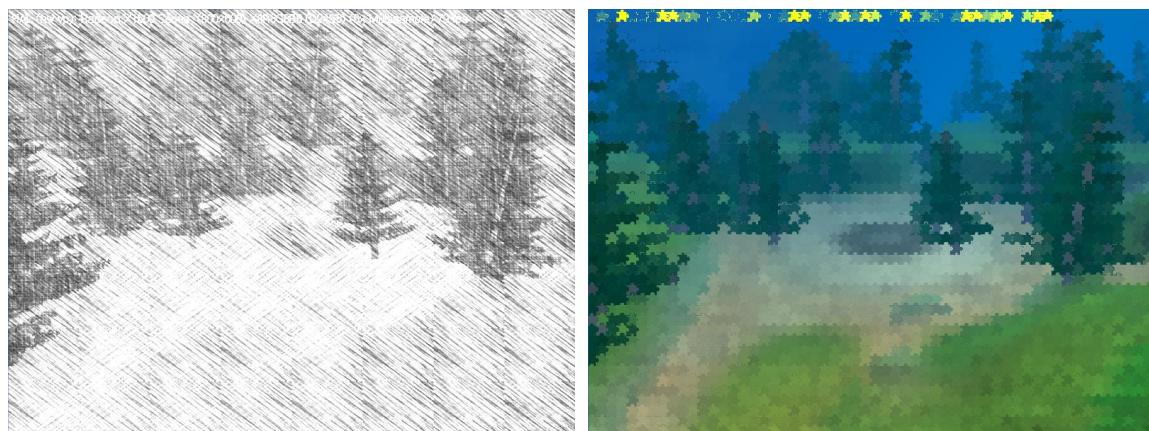
## 6 HLSL – Post Effect

이 장은 앞서 배운 HLSL을 바탕으로 게임에 적용할 만한 쉐이더 코드를 만들어 보는 연습의 장입니다. 정점 쉐이더 보다 픽셀 쉐이더가 양적, 질적으로 가장 많이 게임에서 적용되는 분야이므로 픽셀 처리부분을 집중적으로 연습하겠습니다.

픽셀 쉐이더를 사용하는 부분은 여러 가지가 있겠지만 특히 Post Effect 부분은 같은 장면이라도 전혀 다른 느낌의 연출을 만들 수 있어 게임이 그래픽이 아닌 프로그래머에서 렌더링 품질이 좌우되는 유일한 영역이기도 합니다. 다음 그림을 보면 평범한 장면도 얼마든지 기획의 의도대로 화면을 만들 수 있고 연출에 대한 품질도 개선 시킬 수 있음을 볼 수 있습니다.



<쉐이더 적용 전, 적용 후(Blur-Glare)>



<연필선 효과, 직소>

위의 그림은 게임에서 자주 등장하는 흐림(Blur) 효과 또는 Bloom 효과입니다. 이 것을 스스로 만들 수 있으려면 픽셀 처리가 손에 익어야 합니다. 이 과정에서는 손쉽게 구현할 수 있는 모자이크(Mosaic) 효과를 시작으로 화면 잡음(Noise), 해칭(Hatching) 등을 먼저 연습할 것입니다. 다음으로 흐림(Blur) 효과를 구현할 것입니다. 흐림 효과를 할 수 있다면 포스트 이펙트를 잘 알고 있

다는 것이며 이의 연습으로 크로스 효과도 만들어 보겠습니다. 마지막은 외곽선 추출을 이용해서, 수묵화 효과 등 비 실사 렌더링도 사용해 보겠습니다.

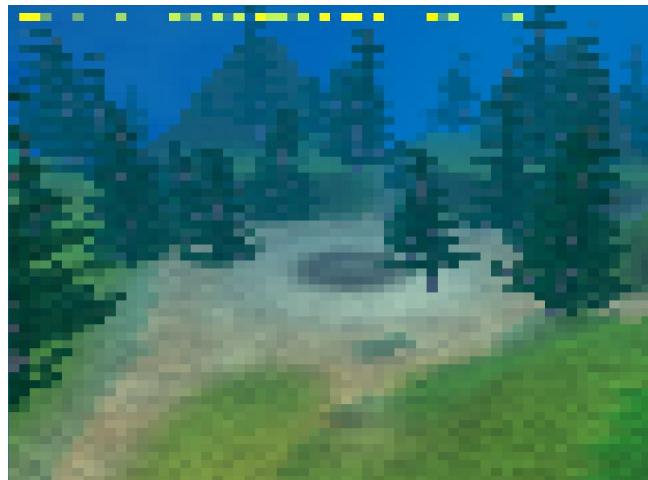
## 6.1 모자이크(Mosaic)

HLSL 이용한 Post Effect의 가장 좋은 연습은 모자이크 효과입니다. 모자이크 효과는 특별한 기술 없이 자와 컴퍼스 연습장 1장만 있으면 기하학적 무늬를 화면 전체에 만들어 낼 수 있습니다. 예를 들어 가며 설명하겠습니다.

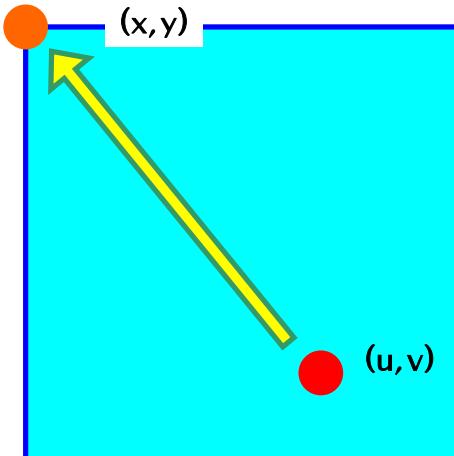
### 6.1.1 직사각형 모자이크

다음 그림은 화면에 정사각형 패턴을 적용한 것입니다. 이것은 아주 간단한 방법으로 먼저 화면 전체를 실시간 텍스처에 저장을 합니다. 이 텍스처의 UV 좌표를 정수형으로 캐스팅합니다. 그러면 소수점 부분이 사라져 버리게 됩니다. 그런데 바로 캐스팅하면 좌표가  $[0, 1]$  범위이기 때문에 전부 0이 됩니다. 따라서 정수를 필요한 만큼 곱한 다음 캐스팅하고 이를 다시 곱한 정수만큼 나누는 것입니다.

예를 들어 사각형의 가로 간격을 10으로 하고 싶으면  $u$  좌표에 10을 곱하고 int 형으로 캐스팅 한 다음 다시 10으로 나눕니다. 그러면 소수점 0.1xxx에서 xxx는 잘려 버리게 됩니다.



<직사각형 모자이크 효과>



다음은 이것을 구현한 쉐이더 코드의 일부입니다.

```
texture m_TxDif;
sampler smpDif = sampler_state
```

```

{
    texture = <m_TxDif>;
    ...
};

struct SvsOut
{
    float4 Pos:POSITION;
    float2 Tex:TEXCOORD0;
};

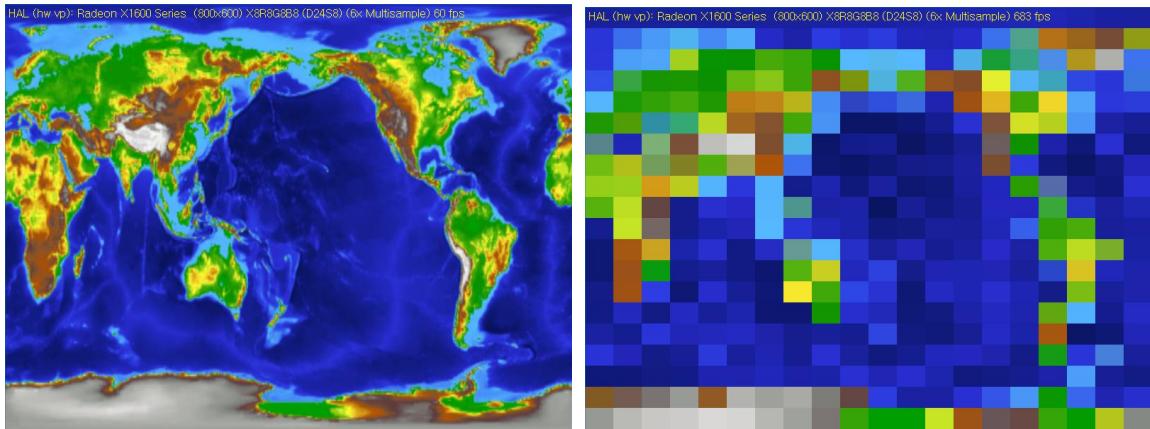
...
float m_TxD=20;           // Image Mosaic Delta
float4 Px1Proc(SvsOut In) : COLOR0
{
    float4 Out=0;
    float u = In.Tex.x;
    float v = In.Tex.y;
    int iX = (int)(u * m_TxD);
    int iY = (int)(v * m_TxD);
    float x = iX/(m_TxD);
    float y = iY/(m_TxD);

    Out = tex2D(smpDif, float2(x,y));
    return Out;
}

```

포스트 이펙트는 앞서 퍽셀 쉐이더에서도 보았듯이 다음 그림처럼 장면을 텍스처에 저장하고 이 텍스처에 퍽셀 쉐이더를 이용해서 마치 전체 장면에 이펙트를 주는 것처럼 만드는 기술입니다.

앞의 쉐이더 코드를 연습하기 위해서 여러분은 이 쉐이더 코드를 올릴 테스트용 프로그램이 필요합니다. 만약 게임 제작 코드에 이 것들을 넣는다면 스파게티 코드가 될 것이라는 점은 분명합니다. 시간이 좀 들더라도 다음과 같은 테스트용 프로그램을 준비합니다.



<쉐이더 테스트용 프로그램: [h4\\_00\\_screen.zip](#)에 직사각형 적용 후>

[h4\\_00\\_screen.zip](#)의 CShaderEx 클래스는 쉐이더 코드를 관리하는 클래스입니다. 실행을 하면 위와 같은 세계지도를 화면에 출력하고 있습니다. 또한 Data 폴더를 보면 위의 그림을 표현하기 위해 기초 쉐이더 코드들이 있습니다. 이 기초 쉐이더 코드 중에서 가장 중요한 것은 Technique에 있는 함수 컴파일 버전을 꾹셀 쉐이더는 "compile ps\_2\_0"으로 즉, 2.0이상으로 버전을 정해야 합니다.

[h4\\_00\\_screen.zip](#)의 "Shader.fx" 파일에 "float m\_Tx0=20" 부분부터 Px1Proc() 함수 안까지 복사해서 붙이고 실행하면 앞의 그림 오른쪽과 같은 그림을 얻을 수 있습니다.

간혹 쉐이더 문법에 다음과 같은 코드를 볼 수 있을 때도 있습니다. 이것은 샘플링을 담당하는 샘플러 객체를 register s3에 지정하는 방법입니다. 참고로 s0, s1, s2, … 는 GPU의 쉐이더 레지스터 고유 이름입니다.

```
sampler smp0:register(s3);
```

이렇게 하면 고정 파이프라인에서 [pDevice->SetTexture(3, "텍스처");] 으로 작성한 3번에 연결된 텍스처를 샘플링 합니다. 또한 사용자가 고정 함수 파이프라인에서 지정한 주소 지정 방식(Address Mode)과 필터링(Filtering) 등을 그대로 사용할 수 있습니다. 이런 방식은 쉐이더에서 제대로 표현되고 있는지를 고정파이프라인과 거의 동일한 환경으로 만들고자 할 때 주로 작성합니다.

위의 구현 후 코드는 [h4\\_01\\_mosaic0\\_rect1.zip](#)에 있습니다.

다음으로 준비할 것은 실제 장면에 쉐이더 코드를 올려봐야 하므로 지형 정도만 구현되어 있는 다음과 같은 코드를 준비합니다.



<쉐이더 적용 대상: [h4\\_00\\_height.zip](#), [h4\\_00\\_height\\_2.zip](#)>

또한 장면을 텍스처에 저장하고 화면에 출력할 수 있도록 코드를 만듭니다. 이 때 [h4\\_00\\_screen.zip](#) 만큼의 코드만 만들어서 실제 장면이 이미지에 출력되는지 코드를 만들어야 합니다. 앞의 오른쪽 그림처럼 나와야 합니다.

[h4\\_00\\_height\\_2.zip](#)의 `CShaderEx` 클래스에는 전체 장면의 픽셀을 전달하는 `SetTexture()` 함수가 추가되어 있습니다. 또한 전체 장면을 텍스처에 저장하기 위해 고정 함수 파일라인으로 구현한 "서피스 효과" 장에 있었던 `IrenderTarget` 객체를 이용했습니다. `IrenderTarget` 객체 방법은 "서피스 효과" 부분을 다시 읽어 보기 바랍니다.

이렇게 해서 우리는 쉐이더 테스트용 프로그램, 장면에 적용해 볼 수 있는 프로그램 두 가지를 준비했습니다. 앞으로 모든 포스트 이펙트에 대한 코드는 쉐이더 테스트용 프로그램에서 먼저 실행해보고, 이후 게임 화면과 유사한 환경에서 이를 다시 실행해 보겠습니다.

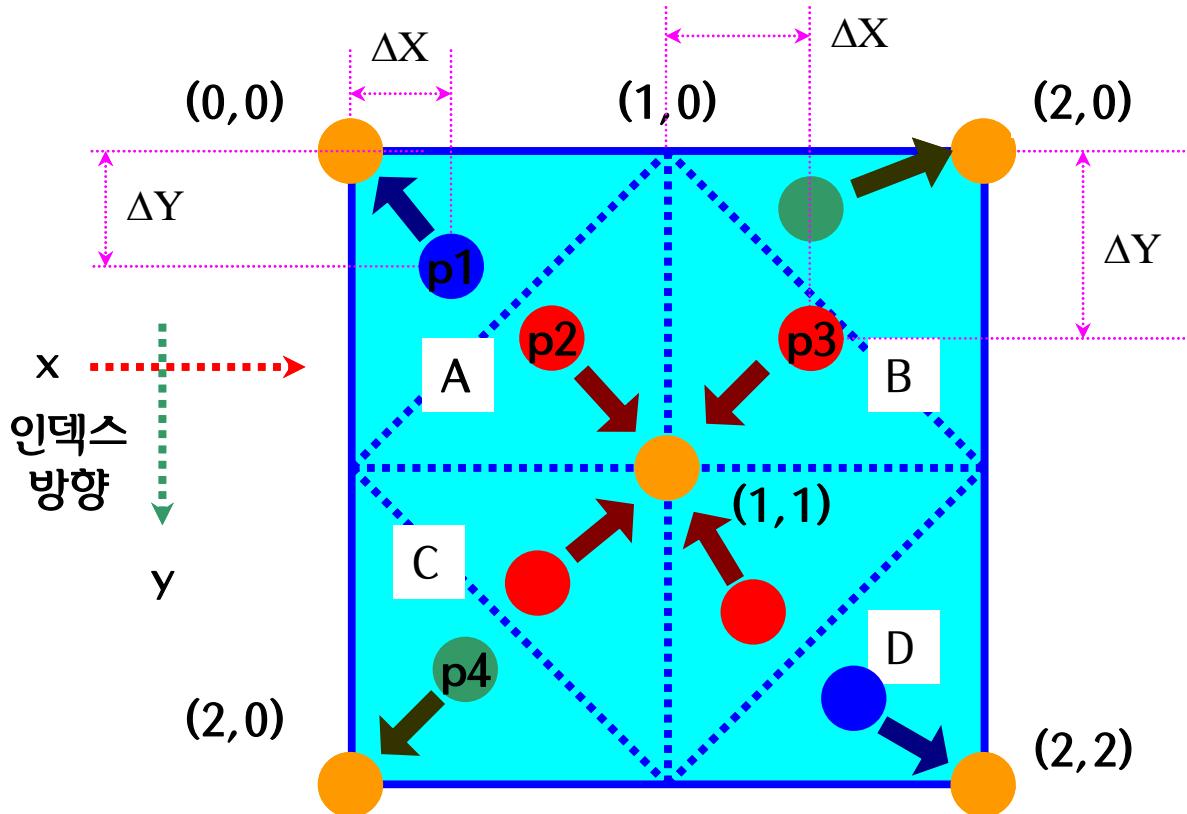
### 6.1.2 마름모형 모자이크

직사각형은 너무나 쉬워서 손이 아직 덜 풀렸을 것입니다. 다음으로 직사각형 보다 약간 난이도 있는 마름모를 적용해 봅시다. 마름모는 의외의 난이도가 있어서 연습장에 한 번 정도는 그려봐야 합니다. (기하학 문제는 그림을 잘 그리면 쉬워집니다.)

그림처럼 먼저 4 구역 A, B, C, D로 설정합니다. 자세히 보면 마름모로 색상을 칠하려면 중심 되는 지점을 먼저 찾아야 되는데 (0,0), (2,0), …, (1,1), (3,1), …, 즉 X+Y가 짝수 인 경우가 중심이 됩니다.

또한 각 점에 대해서 앞서 직사각형에서처럼 곱하기→캐스팅으로 해서 해당 인덱스를 먼저 구합니다. 이 인덱스가 중심인지 아닌지는 modular 연산자 "%"를 이용합니다. "인덱스%2" 하면 짝수, 홀

수 판정이 나오고 둘 다 (0,0) 또는 (1,1)의 경우에는 인덱스가 중심 좌표가 됩니다.



<마름모형 UV인덱스>

또한 인덱스에서 거리 = 자신위치- 중심위치로  $\Delta X$ ,  $\Delta Y$ 를 구합니다. 이것을 가지고 완전한 중심 인덱스를 찾습니다.

예를 들어 점 p1, p2는 인덱스가(0,0) 입니다. 이들의 중심점은 변동 없이 (0,0)을 처음에 정한 인덱스를 중심 인덱스로 합니다. 점 p1의 경우는  $\Delta X + \Delta Y < 1$  가 됩니다. 그런데 점 p2는  $\Delta X + \Delta Y > 1$  이 됩니다. 따라서 점 p2는 x 방향으로 인덱스 x + 1, y 방향으로 인덱스 y + 1로 각각 1만큼 올립니다.

이번에는 점 p3를 봅시다. 이 점의 인덱스는 (1,0)을 이 점은 중심 점이 아닙니다. 따라서 x 방향 또는 y 방향으로 중심을 이동해야 하는데 x 방향으로는 앞서 구한 방법대로  $\Delta X$ 를 그대로 구하고 y 방향은 1을 더한 만큼에서  $\Delta Y$ 를 구합니다. 만약  $\Delta X + \Delta Y < 1$  이면 y 방향으로 1만큼 증가한 값이 중심 인덱스이고 그렇지 않으면 x 방향으로 1만큼 올린 값이 중심인덱스가 됩니다.

이와 같은 방법으로 다음과 같은 코드를 만들어 낼 수 있습니다.

```
float4 Px1Proc(SvsOut In) : COLOR0
```

```
...
```

```

int      mX = nX%2;
int      mY = nY%2;

float DelX = 0; float DelY = 0; float Del = 0;

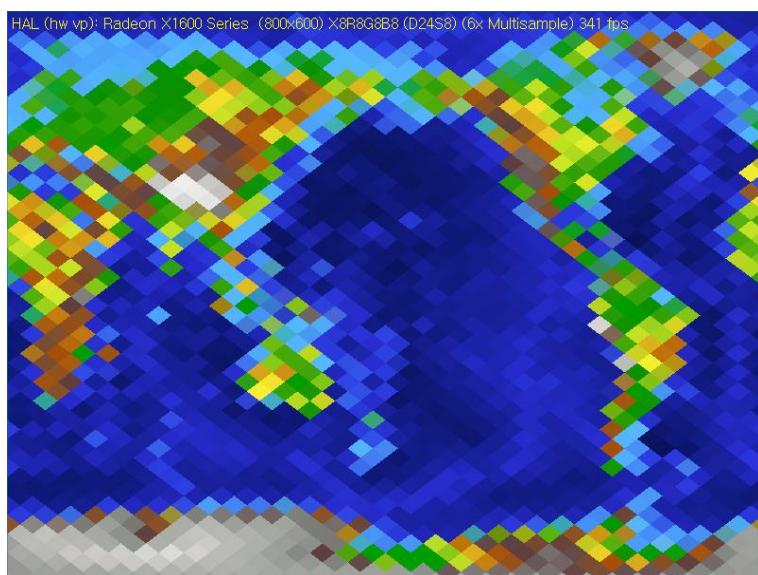
if( (0==mX && 0== mY) || (1==mX && 1== mY))
{
    DelX = TxX - nX;
    DelY = TxY - nY;
    Del = DelX + DelY;

    if(Del<1)
    {
        U = nX;      V = nY;
    }
    else
    {
        U = nX + 1.f;
        V = nY + 1.f;
    }
}
else
{
    DelX = TxX - nX;
    DelY = nY+1 - TxY;
    Del = DelX + DelY;

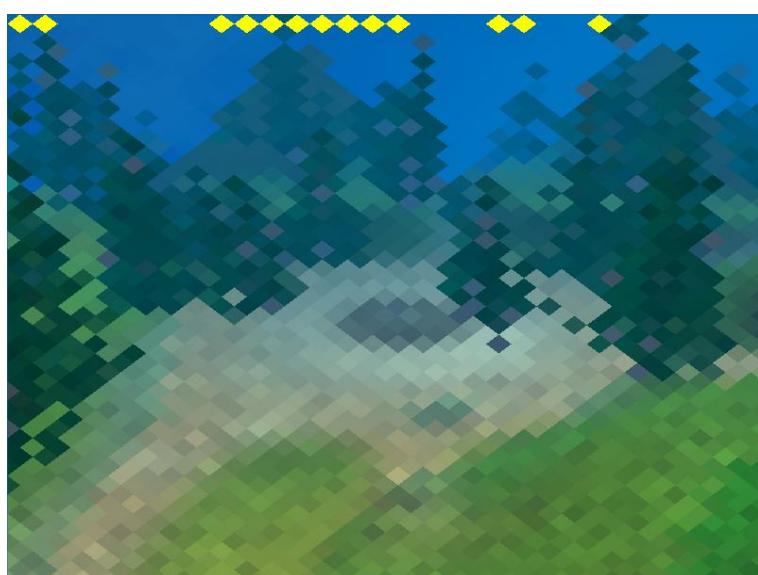
    if(Del<1)
    {
        U = nX;
        V = nY + 1.f;
    }
    else
    {
        U = nX + 1.f;
        V = nY;
    }
}

```

```
    }  
  
    U /= m_fRpt;  
    V /= m_fRpt;  
  
    Out = tex2D(smpDiff, float2(U, V));  
    return Out;  
}
```



<파일명: [h4\\_01\\_mosaic1\\_lozenge.zip](#)>



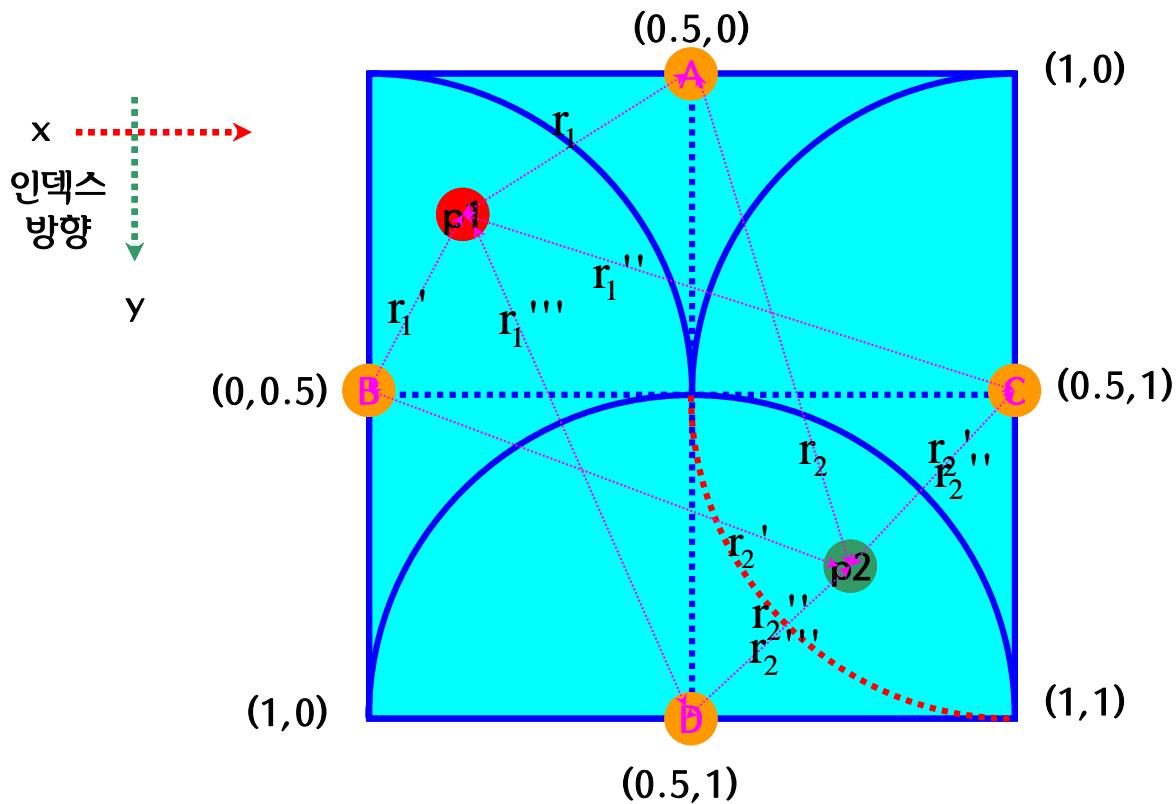
<파일명: [h4\\_01\\_mosaic1\\_lozenge\\_a.zip](#)>

전체 코드는 [h4\\_01\\_mosaic1\\_lozenge.zip](#) 와 [h4\\_01\\_mosaic1\\_lozenge\\_a.zip](#) 의 "data" 폴더의

"shader.fx" 파일을 참고 하기 바랍니다.

### 6.1.3 은행잎 모자이크 1

마름모 형태는 쉐이더 코드보다 기하학적 패턴 만들기에 더 노력이 들었습니다. 지금 하고자 하는 은행잎 모양도 마름모와 비슷하게 코드를 만들기 보다는 그림을 그리고 해석하는 데 시간이 더 많이 듭니다. 이런 패턴 만드는 일을 계속 하다 보면 포스트 이펙트에서 꽈셀 쉐이더의 역할이 선명해 질것입니다.



<은행잎 패턴>

그림을 보면 x,y인덱스의 범위를 [0, 1]로 만들 때 이 정사각형 안의 u, v위치는 황색 점 A, B, C, D들 중 하나에서 샘플링 해야 합니다. 가장 편하게 만들 수 있는 코드는 점들을 돌아가면서 샘플링 하되 거리를 비교해서 거리가 0.5(1의 반지름) 안에 있을 때만 샘플링 하는 것입니다.

예를 들어 점 p1은 A와 거리를 구합니다. 구한 거리는 그림에서 보듯 0.5가 안되어 샘플링을 합니다. 다음 점 B와 비교하면 거리가 0.5보다 작습니다. B의 위치에서 샘플링하고 이전 값을 바꿉니다. 다음 점 C와 비교해보면 거리가 0.5를 넘게 되어 C에서는 샘플링을 안 합니다. 마지막 D도 마

찬가지로 거리가 0.5보다 커서 샘플링을 안 합니다. 이렇게 되면 p1의 색상은 점 A가 됩니다. 또 다른 점 p2를 봅시다. p2는 A, B 점에 대해서는 거리가 0.5보다 커서 샘플링을 안 합니다. 점 C와 거리는 0.5 미만이므로 샘플링을 합니다. 그런데 점 D와도 거리가 0.5 미만이라 최종 점 D에서 샘플링 한 색상을 최종 색상으로 정합니다.

코드의 구현에서는 거리로 비교하지 않고 거리의 제곱으로 비교합니다. 따라서 0.5가 아닌 0.25로 비교합니다. 또한 색상은 어떤 식으로든 결정이 되어야 하므로 최초 점 A에서 무조건 샘플링을하게 합니다. 이렇게 되면 중간에 색상을 잃어버리는 일을 없어 집니다.

이 내용을 가지고 쉐이더 코드를 구현하면 다음과 같습니다.

```
float m_fRpt; // Image Repeat
float4 Px1Proc(SvsOut In) : COLOR0
...
float2 Tx = In.Tex * m_fRpt;
float2 Del=0;
float2 t=0;

int iX = (int)(In.Tex.x * m_fRpt);
int iY = (int)(In.Tex.y * m_fRpt);

// 처음 시작(0.5, 0)
t = float2(0.5 + iX, 0 + iY);
t /= m_fRpt;
Out = tex2D( smpDif, t);

// (0, 0.5)
t = float2(0.0 + iX, 0.5 + iY);
Del = Tx - t;

if( Del.x*Del.x + Del.y*Del.y<=0.25)
{
    t /= m_fRpt;
    Out = tex2D( smpDif, t);
}

// (1, 0.5)
```

```

t    = float2(1.0 + iX, 0.5 + iY);
De1 = Tx - t;

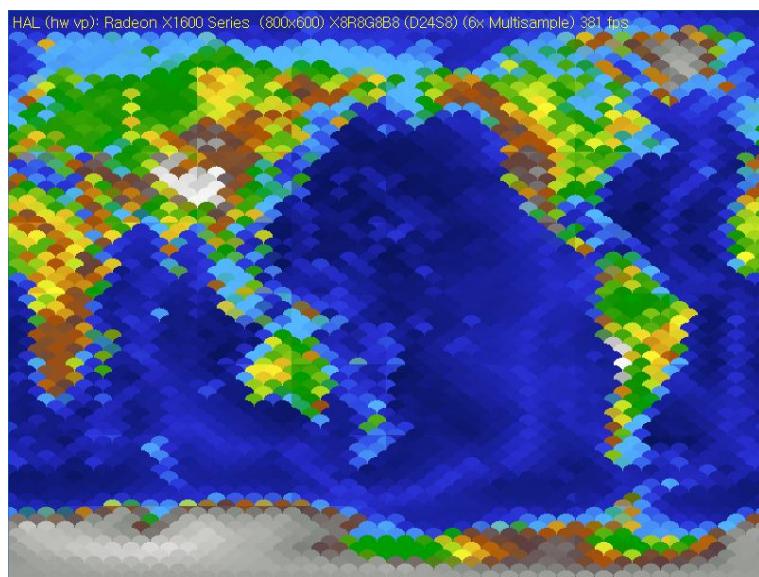
if( De1.x*De1.x + De1.y*De1.y<=0.25)
{
    t /= m_fRpt;
    Out = tex2D( smpDif, t );
}

// (0.5, 1)
t    = float2(0.5 + iX, 1. + iY);
De1 = Tx - t;

if( De1.x*De1.x + De1.y*De1.y<=0.25)
{
    t /= m_fRpt;
    Out = tex2D( smpDif, t );
}

return Out;
}

```



<은행잎 패턴: [h4\\_01\\_mosaic1\\_ginkgo.zip](#)



<은행잎 패턴: [h4\\_01\\_mosaic1\\_ginkgo\\_a.zip](#)>

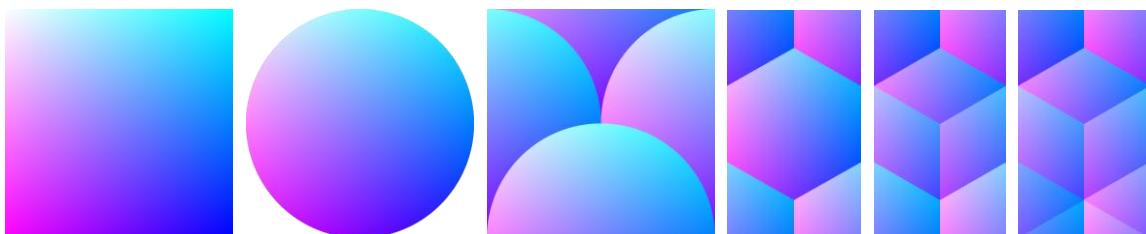
전체 코드는 [h4\\_01\\_mosaic1\\_ginkgo.zip](#), [h4\\_01\\_mosaic1\\_ginkgo\\_a.zip](#)을 참고하기 바랍니다.

#### 6.1.4 은행잎 모자이크 2

직사각형, 마름모, 은행잎 형태는 손으로 계산할 수 있어서 프로그램이 가능합니다. 그런데 이렇게 직접 패턴에 대한 구현을 쉐이더 코드로 만드는 것은 많은 부담이 아닐 수 없습니다.

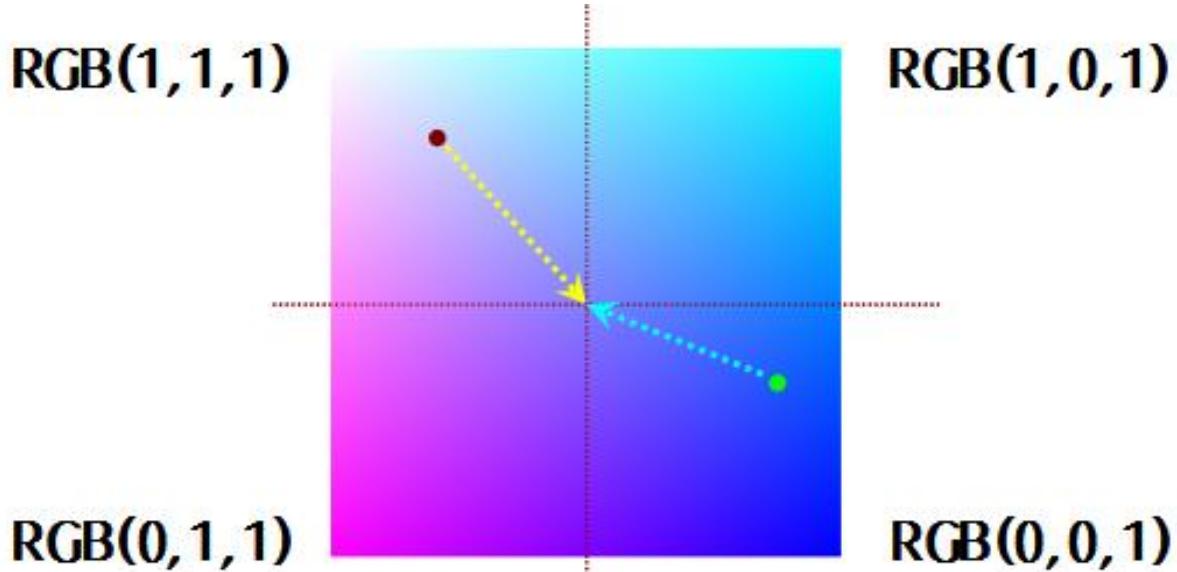
만약 그래픽으로 만든 패턴이 있으면 프로그램은 패턴 이미지에 저장된 픽셀을 새로운 샘플링 좌표  $u$ ,  $v$ 로 이  $u$ ,  $v$ 로 바꾸고 화면에 출력할 텍스처의 픽셀을 샘플링 한다면 프로그램은 상당히 간소화 될 것이라는 것은 분명합니다.

다음 그림은 그래픽 패턴입니다.



<그래픽으로 작성한 패턴>

이 그래픽 패턴 중에서 가장 왼쪽에 있는 사각형을 보기 바랍니다. 이 사각형을 자세히 보면 다음과 같이 색상이 분포되어 있음을 할 수 있습니다.



&lt;패턴 맵1&gt;

왼쪽 상단은  $\text{RGB}(255, 255, 255)$ 입니다. 쉐이더에서 색상은  $[0, 1]$  범위로 사용하고 있으므로 이 범위 값으로 색상으로 바꾸면  $\text{RGB}(1, 1, 1)$ 이 됩니다. 마찬가지로 우측하단도  $\text{RGB}(0, 0, 255)$ 로  $\text{RGB}(0, 0, 1)$ 이 됩니다. 만약 그림의 갈색 점이나 초록색 점을 얻은 색상을  $u, v$ 로 사용한다면 이를 점은  $\text{RGB}(128, 128, 255) \rightarrow \text{RGB}(0.5, 0.5, 1)$ 로 옮겨야 정확한 중심이 됩니다.

초록 점 또는 갈색 점의 색상  $R, G$ 를  $\text{Pattern}(r, g)$  라 하면 중심 색으로 옮기는 이동 크기는  $\text{Pattern}(r-0.5, g-0.5)$ 가 됩니다.

따라서 새로운  $U, V$  샘플링 좌표는 다음과 같이 결정할 수 있습니다.

$$\text{Old}(U, V) = \text{기준}(U, V)$$

$$\begin{aligned} \text{New}(U, V) &= \text{Old}(U, V) + \text{중심점으로 이동} \\ &= \text{Old}(U, V) + \text{Pattern}(r-0.5, g-0.5) \end{aligned}$$

만약 반복 횟수가 있으면 이 반복 횟수만큼 늘린 다음 줄입니다.

$$\text{New.U} = (\text{Old.U} * \text{RepeatX} - 0.5) / \text{RepeatX}$$

$$\text{New.V} = (\text{Old.V} * \text{RepeatY} - 0.5) / \text{RepeatY}$$

이에 대한 쉐이더 구현은 [h4\\_01\\_mosaic2\\_ginkgo.zip](#) 의 "Shader.fx"에 구현되어있습니다.

```
texture m_TxPtn;
sampler smpPtn = sampler_state
```

```

{
    texture = <m_TxPtn>;
    ...
};

static float    fRepeatX = 4;
static float    fRepeatY = 3;
float   m_fRpt=4;           // Image Repeat

float4 Px1Proc(SvsOut In) : COLOR0
{
    ...
    float2 TxOld = 0;
    float2 TxBias = 0;

    fRepeatX *= m_fRpt;
    fRepeatY *= m_fRpt;

    TxOld.x = In.Tex.x * fRepeatX;
    TxOld.y = In.Tex.y * fRepeatY;

    t1 = tex2D(smpPtn, TxOld);

    // 텍스처 좌표를 이동
    TxNew.x = (t1.x - 0.5f)/fRepeatX;
    TxNew.y = (t1.y - 0.5f)/fRepeatY;

    TxNew += In.Tex;

    t0 = tex2D(smpDif, TxNew);

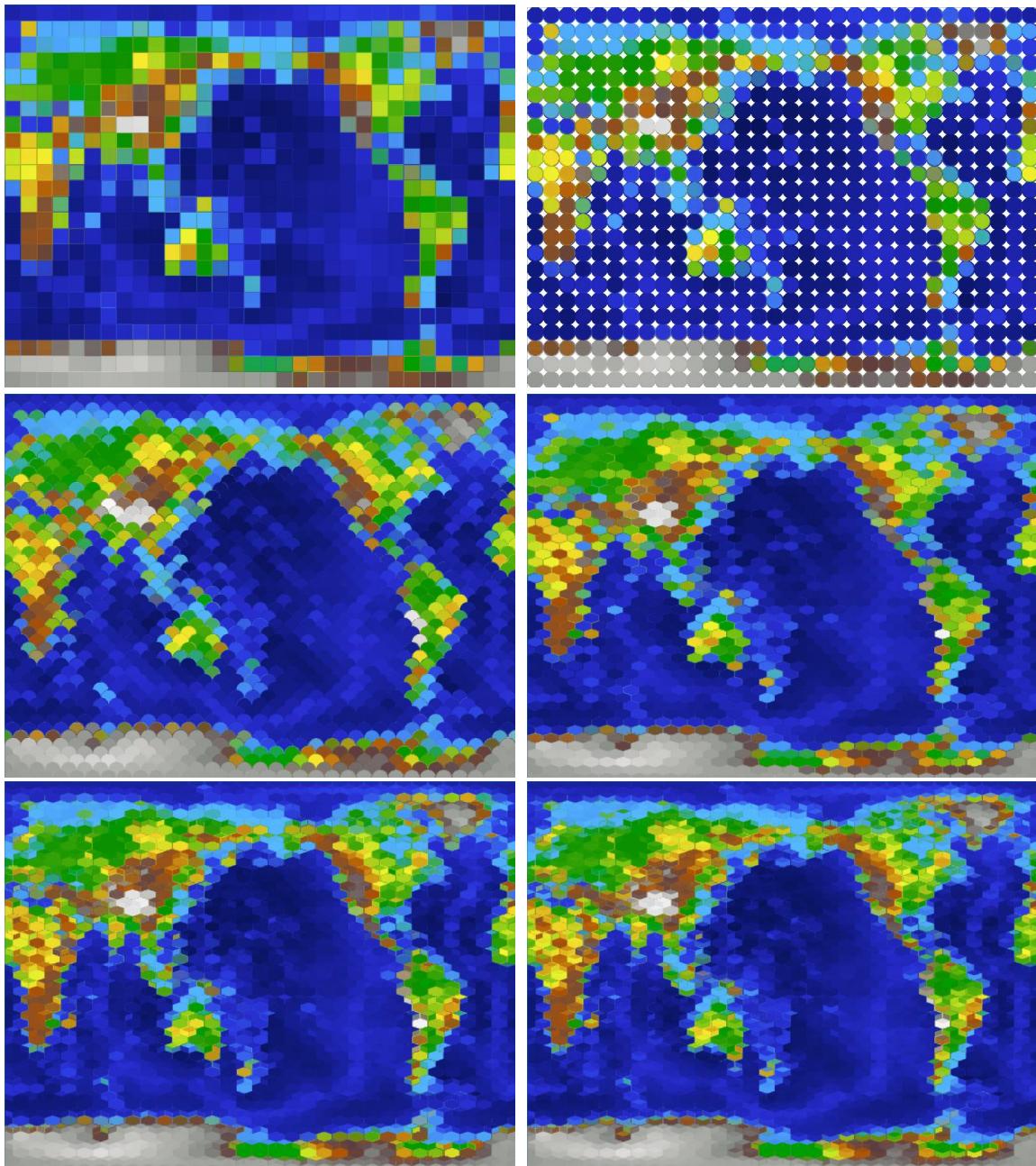
    if(t1.x>0.99 && t1.y >0.99)
        Out= 1;
    else
        Out = t0;

    return Out;
}

```

}

패턴 텍스처의 색상을 가져오기 위해서 sampler smpPtn가 추가 되었습니다. 또한 패턴의 크기가 화면의 가로 세로 비율을 맞추기 위해 X 방향, Y 방향으로 4, 3을 반복 회수에 곱해서 사용합니다. 다음 그림은 이 패턴 맵을 적용한 화면입니다.

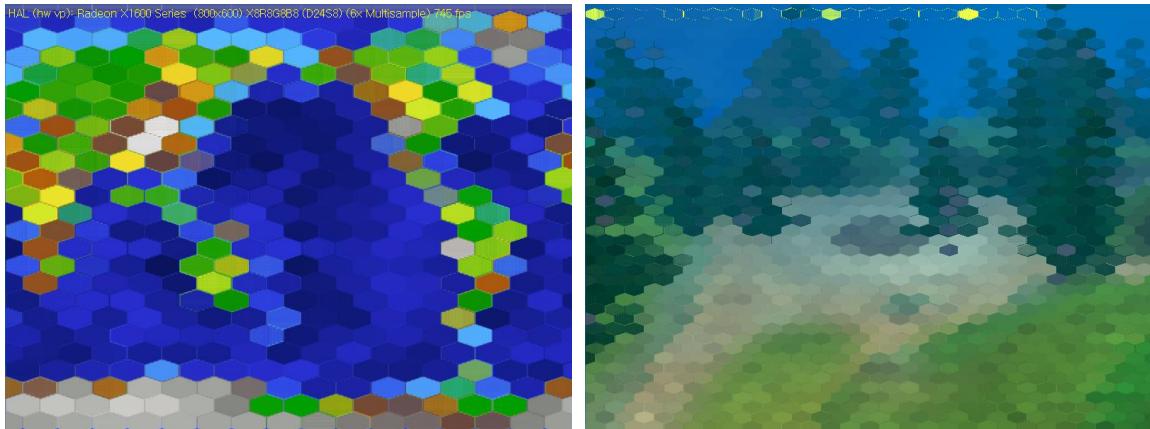


<패턴 맵2 - 쉐이더 시험 화면>



<패턴 맵3 - 게임 화면>

만약 두 개의 장면을 교차하는 연출을 만들 때 이 패턴 반복 횟수를 동적으로 변화시 적용해 볼 수 있습니다. 이것을 실제 화면에 적용해 보면 좋겠지만 예전상 여기서는 그런 것이 있다고 가정하고 하나의 화면에 대해서만 적용하면 다음과 같은 화면을 얻을 수 있습니다.

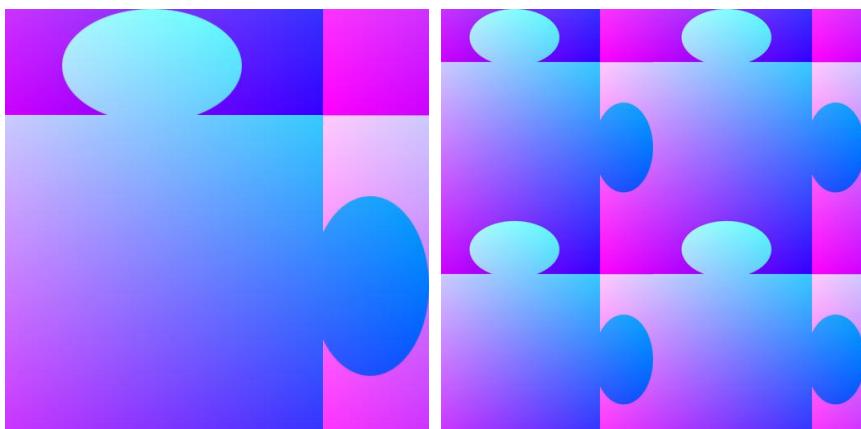


<반복 횟수를 변환한 패턴: [h4\\_01\\_mosaic2\\_hexa.zip](#), [h4\\_01\\_mosaic2\\_hexa\\_a.zip](#)>

### 6.1.5 직소(Jigsaw) 퍼즐

패턴의 이미지는 타일처럼 색상의 변화가 연속이어야 합니다. 또한 그 색상들은 중심 색을 정확하게 샘플링 할 수 있어야 합니다. 지금까지 화면에서 사용한 이미지 패턴들은 대칭이 존재해서 만들기가 어렵지 않았습니다. 그런데 직소는 대칭보다 비대칭에 가깝습니다. 따라서 이전에 해왔던 작업보다 노력이 많이 필요한데 여러분들도 이와 비슷한 것을 주제 삼아 만들어 보기 바랍니다.

처음에는 화가 에셔(M.C. Escher)의 그림을 모티브로 비대칭 패턴을 만들려고 했으나 실패하고 간단한(사실 간단하지 않은) 직소를 재미 삼아 도전해 보았습니다. 다음 그림은 직소원본과 이 그림을 연속으로 배치 했을 때의 모습입니다.



<직소 패턴>

이 직소 모양도 쉐이더 코드는 이전과 거의 같습니다. 단지 차이는 전에는 사각형의 패턴의 중심 색을 샘플링 하도록 했지만 이 직소는 x 방향의  $4/5$  되는 지점에서는 오른쪽 방향으로 0.25만큼 더 이동해야 합니다. y 방향은  $1/5$  되는 지점에서 위로 0.5만큼 더 이동해야 합니다.

이것을 의사 코드(pseudo-code)로 만들면

샘플링 텍스처 좌표  $t(x, y) = \text{Pattern}(\text{원본}(U, V))$

새로운 좌표  $U = t.x * 1.25 - 0.75$

새로운 좌표  $V = t.y * 1.25 - 0.50$

이 됩니다.

예제의 "shader.fx" 파일에는 반복 횟수까지 고려해서 다음과 같이 구현되어 있습니다.

```
float4 PxlProc(SvsOut In) : COLORO
...
TxOld.x = In.Tex.x * fRepeatX;
TxOld.y = In.Tex.y * fRepeatY;

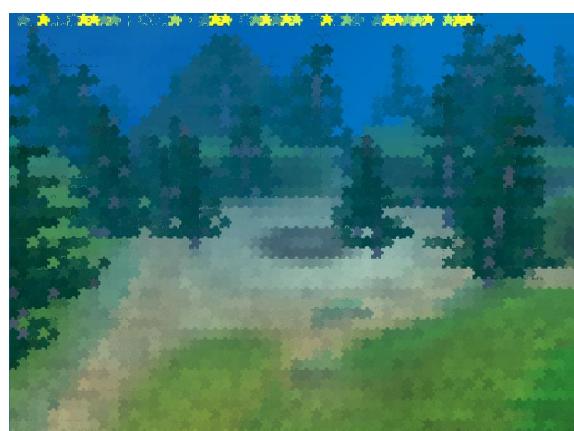
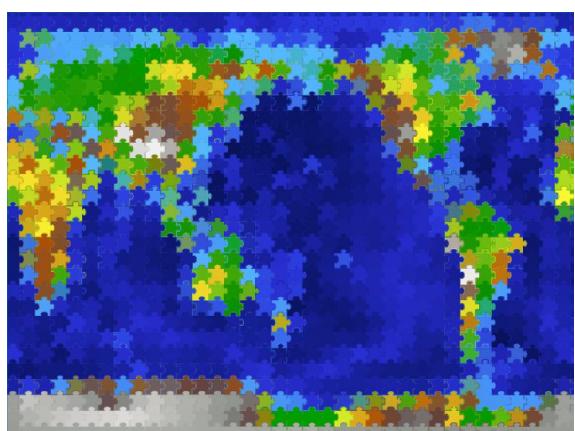
t1 = tex2D(smpPtn, TxOld);

// 텍스처 좌표를 이동
TxNew.x = (t1.x * 1.25f - 0.75f)/fRepeatX;
TxNew.y = (t1.y * 1.25f - 0.50f)/fRepeatY;

TxNew += In.Tex;

t0 = tex2D(smpDiff, TxNew);
...
```

다음 두 그림은 직소 패턴을 이용해서 화면입니다.



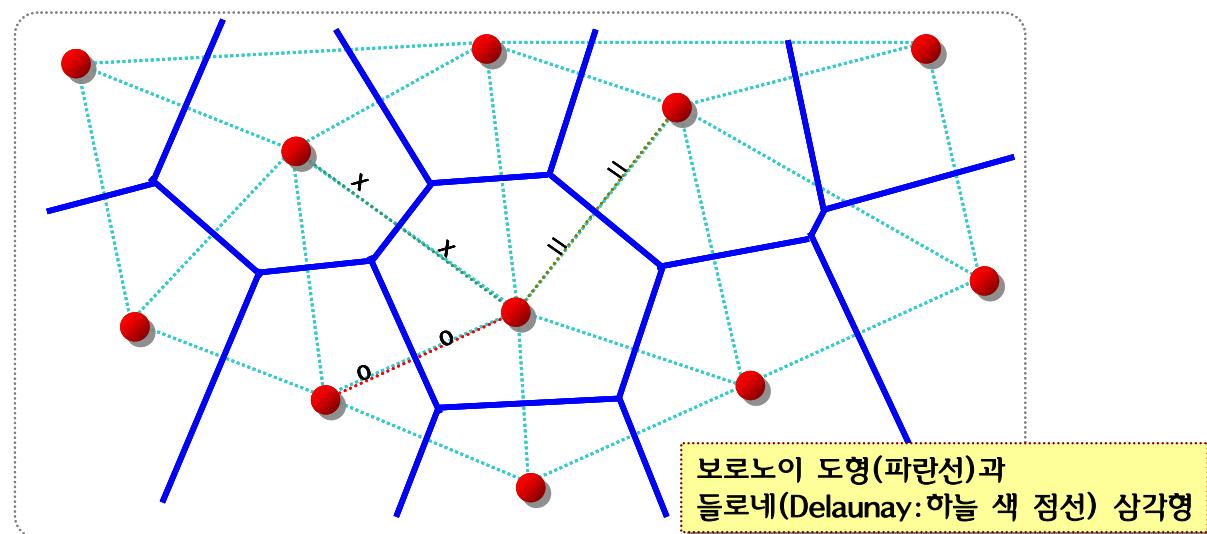
<작소 패턴: [h4\\_01\\_mosaic2\\_jigsaw.zip](#), [h4\\_01\\_mosaic2\\_jigsaw\\_a.zip](#)>

### 6.1.6 Voronoi Diagram

지금까지 만든 패턴을 적용해서 만든 포스트 이펙트는 Non Reality에 가깝습니다. 이것을 3D에서 NPR(Non Photo-realistic Rendering)이라 합니다. NPR 중에서 화면 자체를 유화(Oil Paint), 종이 찢어 붙이기, 중세 교회의 모자이크, 스테인드 글라스(Stained Glass) 등과 같은 효과를 만들기 위해서 가장 필요한 것이 화면의 영역을 분할 하는 것입니다. 이 때 보로노이 도형(Voronoi Diagram)을 이용합니다.

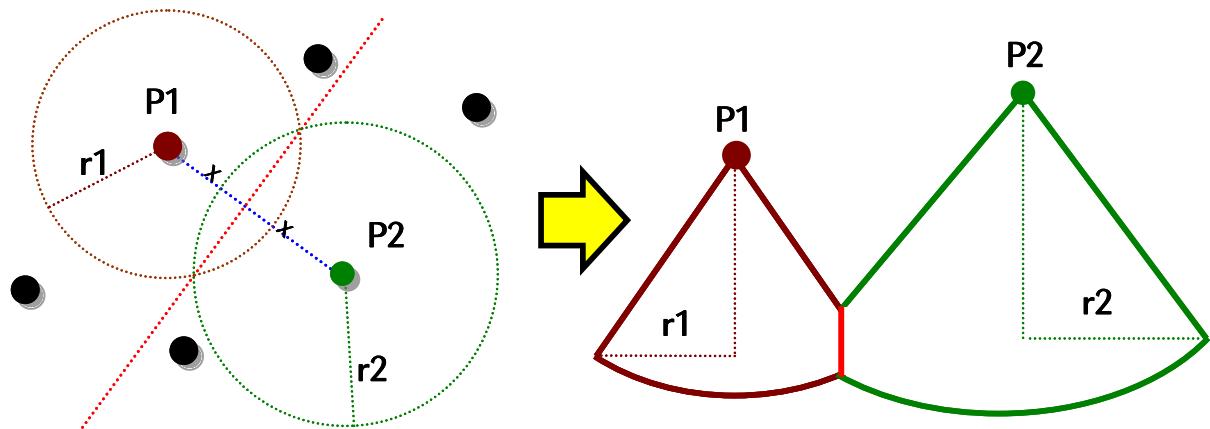
이 도형은 다음 그림처럼 인접한 점들과 정확히 절반 되는 위치에서 수직선을 그어 이 선들이 만나는 지점들을 꼭지점으로 다각형 입니다. 보로노이 도형을 만들면 들로네 삼각형(Delaunay Triangle)을 구성할 수 있어서 정점을 가지고 적절한 삼각형 메쉬를 구성하는 데 사용됩니다. 보로노이가 영역에 대한 문제이다 보니 지형, 사회적인 서비스 등에서도 많이 이용되는 도형이니 자료를 찾아 보기 바랍니다.

이 도형을 Adobe Photoshop에서도 볼 수 있는데 이미지를 열고 Filter → Pixelate → Crystallize를 선택하면 이 도형으로 만들어진 이미지를 얻을 수 있습니다.



보통 보로노이 도형은 가장 왼쪽에 있는 점부터 이 점에 가장 가까운 점들부터 수직선을 그어나가면서 선들끼리 만나는 점을 계속 갱신해가면서 맨 오른쪽의 마지막 점까지 이를 진행 합니다. 그런데 여기서 우리는 색상을 보로노이 도형에 맞게 채우는 것이 목적이므로 이 알고리즘을 이용하지 않고 다음과 같이 원뿔을 이용할 것입니다.

보로노이 도형을 위해서 보지 않고 옆에서 보면 수직선은 적당히 큰 반경을 가진 원뿔의 경계가 됨을 쉽게 알 수 있습니다.



<보로노이 도형: 원뿔의 인접 면으로 해석>

즉 처음부터 색상을 가진 3차원 원뿔을 임의로 화면에 연출하면 색상이 채워진 보로노이 도형을 만들 수 있게 됩니다.

이 때 원뿔의 색상은 화면을 저장한 픽셀에서 가져와야 하는데 원뿔의 위치를 픽셀의 U,V로 고정시키는 것입니다. 이렇게 되면 원뿔의 위치로 구한 픽셀의 색상이 하나의 원뿔 전체에 같은 색으로 결정이 될 것입니다. 따라서 쉐이더 코드는 정점의 텍스처 좌표에서 샘플링이 아닌 외부에서 주어진 위치에 의해 샘플링 되도록 다음과 같이 만들어야 합니다.

```
uniform float2 m_UV;

float4 Px1Proc(SvsOut In) : COLOR0
{
    float4 Out=0;
    float4 t0=0;

    t0 = tex2D(smpDiff, m_UV);
    Out = t0;
    return Out;
}
```

복잡할 것 같았는데 코드가 단순하죠?

대신 렌더링에서는 다음과 같이 원뿔들을 보로노이 도형을 구성하는 점들만큼 렌더링 해야 합니다.

```

void CShaderEx::Render()
{
    ...
    m_pEft->SetTexture("m_TxDif", m_pTex);

    for( int i=0; i<m_nCon; ++i)
    {
        // 위치는 [-1.1] 이었으므로 범위를 [0,1]한다.
        uv = ( m_pPos[i] + D3DXVECTOR4(1,1,0,0) ) * 0.5;
        uv.y = 1 - uv.y;

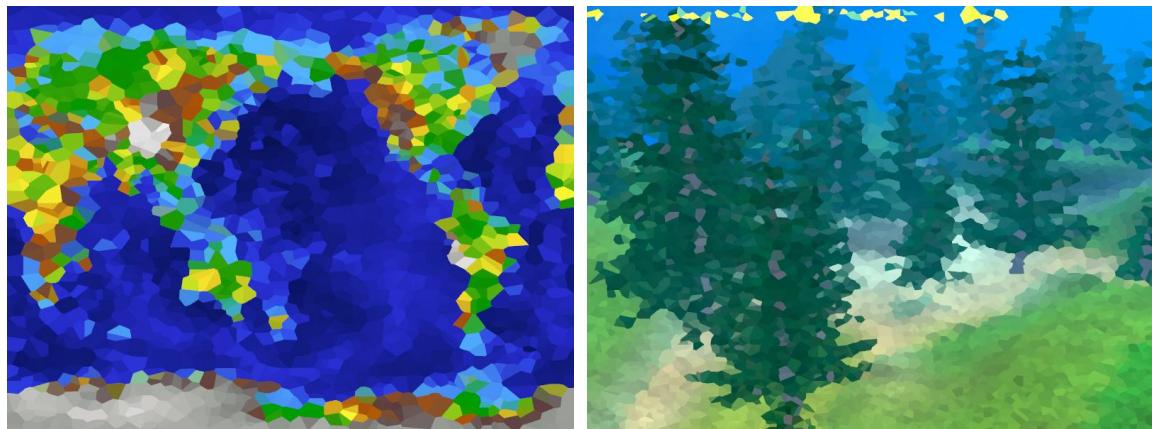
        m_pEft->SetVector("m_UV", &uv);

        mtI._41 = m_pPos[i].x;
        mtI._42 = m_pPos[i].y;

        m_pDev->SetTransform(D3DTS_WORLD, &mtI);
        m_pCon->DrawSubset(0);
    }
    ...
}

```

다음 그림은 보로노이 도형을 출력한 화면입니다.



<보로노이 도형: [h4\\_01\\_mosaic3\\_voronoi.zip](#), [h4\\_01\\_mosaic3\\_voronoi\\_a.zip](#)>

과제) 게임에서 적용 될 수 있는 보로노이 도형과 NPR에 대한 자료를 정리해 오시오.

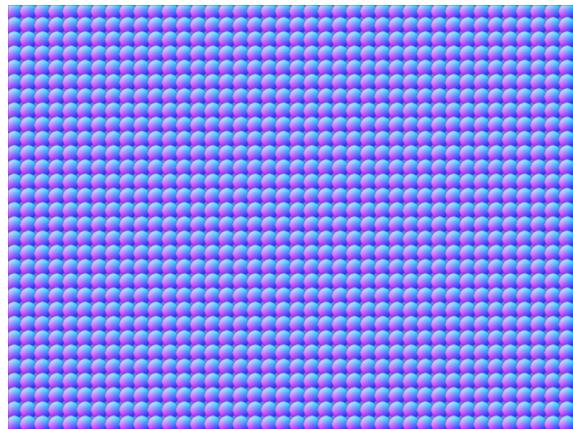
## 6.2 화면 섭동 (Perturbation)

빛은 방안으로 들어와도 한쪽을 보지 못하게 특정한 형태로 유통불통한 모습한 무늬 유리(Embossed Glass)를 본 적이 있을 것입니다. 또한 뜨거운 여름의 아스팔트 위, 장작 불의 열기, 봄철 기온차이로 아지랑이 효과도 본 적이 있을 것입니다. 이러한 자연 현상은 빛의 굴절에 의해 발생합니다. 그런데 이것을 3D로 해석을 하면 전체 장면을 구성하는 픽셀을 특정한 형태를 통해서 움직이는 것으로 볼 수 있습니다. 또한 픽셀을 움직이게 한다는 것은 샘플링의 U, V 좌표를 변동하는 것과 같은 의미가 됩니다.

이 장에서는 무늬 유리, 파티클(Particle)을 이용해서 이러한 효과를 구현해 보겠습니다.

### 6.2.1 무늬 유리 효과(Embossed Glass) 1

무늬 유리는 일정한 무늬가 올록볼록 한 형태로 배치되어 있는 유리입니다. 이 형태를 픽셀의 움직이게 하는 요소로 본다면 우리는 무늬 유리를 3D로 구현하기 위해서 전체 장면의 픽셀을 움직일 다음과 같은 이미지를 생각할 수 있습니다.



<Embossing 텍스처>

그런데 이 올록볼록한 텍스처 이미지를 어떻게 구성할 것인가 잘 생각해 보면 이전에 원의 형태로 은행잎을 만들 때 쓴 텍스처를 이용해서 만들 수 있음을 알 수 있습니다. 즉, Embossing 텍스처를 실시간으로 먼저 만들고 이 Embossing 텍스처에 원을 그릴 때 사용하던 이미지를 적당한 크기와 간격으로 렌더링 하면 Embossing 텍스처는 우리가 원하는 형태의 무늬 유리가 됩니다.

그리고 이 Embossing 텍스처를 앞서 패턴을 사용할 때처럼 픽셀을 움직이게 하는 용도로 사용하면 우리는 전체 장면에 무늬 유리 효과를 만들어 낼 수 있게 됩니다.

[h4\\_02\\_emboss0.zip](#) 의 INT CShaderEx::Restore() 함수에는 다음과 같이 Embossing 텍스처를 만드

는 코드가 있습니다.

```
IrenderTarget* m_pTexP; // Embossing Texture
...
m_pTexP->BeginScene();

m_pDev->Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER, 0xFF8080FF, 1, 0);
...
m_pDev->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
...
m_pDev->SetRenderState(D3DRS_ZENABLE, FALSE);
m_pDev->SetFVF(D3DFVF_XYZRHW | D3DFVF_TEX1);

m_fW = 12;

struct T
{
    D3DXVECTOR4 p;
    D3DXVECTOR2 t;
};

T pVtx[4];

pVtx[0].p = D3DXVECTOR4(0,0,0,1);
pVtx[1].p = D3DXVECTOR4(0,0,0,1);
pVtx[2].p = D3DXVECTOR4(0,0,0,1);
pVtx[3].p = D3DXVECTOR4(0,0,0,1);

pVtx[0].t = D3DXVECTOR2(0,0);
pVtx[1].t = D3DXVECTOR2(1,0);
pVtx[2].t = D3DXVECTOR2(1,1);
pVtx[3].t = D3DXVECTOR2(0,1);

INT i, j;
for(j=0; j<31; ++j)
{
    for(i=0; i<41; ++i)
```

```

{
    pVtx[0].p.x = i*10 - m_fW;
    pVtx[0].p.y = j*10 - m_fW;

    pVtx[1].p.x = pVtx[0].p.x + m_fW;
    pVtx[1].p.y = pVtx[0].p.y + 0 ;

    pVtx[2].p.x = pVtx[0].p.x + m_fW;
    pVtx[2].p.y = pVtx[0].p.y + m_fW;

    pVtx[3].p.x = pVtx[0].p.x + 0 ;
    pVtx[3].p.y = pVtx[0].p.y + m_fW;

    pVtx[0].p *= 2; pVtx[1].p *= 2;
    pVtx[2].p *= 2; pVtx[3].p *= 2;

    m_pDev->SetTexture(0, m_pTex1);
    m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(T));
}

...
m_pTexP->EndScene();

```

다 아는 내용이라 간단히 설명 생략하겠습니다. Clear() 함수에서 0xFF8080FF 값을 RGBA(0.5, 0.5, 1, 1)에 해당합니다. 즉 픽셀의 움직임을 0으로 한 것입니다. 중간에 구조체를 선언하고 정점 4개의 U, V를 정하고 for 문을 돌면서 위치만 설정해서 렌더링 하는 부분은 Embossing 형태를 만드는 것입니다. 디바이스의 SetTexture() 함수에 사용한 텍스처는 원을 패턴으로 그릴 때 사용한 텍스처입니다.

쉐이더 코드는 다음과 같이 처음 패턴을 시작할 때의 모습 그대로 돌아왔습니다.

```

float m_fWidth;

float4 Px1Proc(SvsOut In) : COLOR0
{
    float4 Out=0;
    float4 t0=0;
    float4 t1=0;

```

```

float2 TxOld = 0;
float2 TxNew = 0;

TxOld.x = In.Tex.x;
TxOld.y = In.Tex.y;

t1 = tex2D(smpPtn, TxOld);

// 텍스처 좌표를 이동
TxNew.x = (t1.x - 0.5f)/m_fWidth;
TxNew.y = (t1.y - 0.5f)/m_fWidth * 4./3;

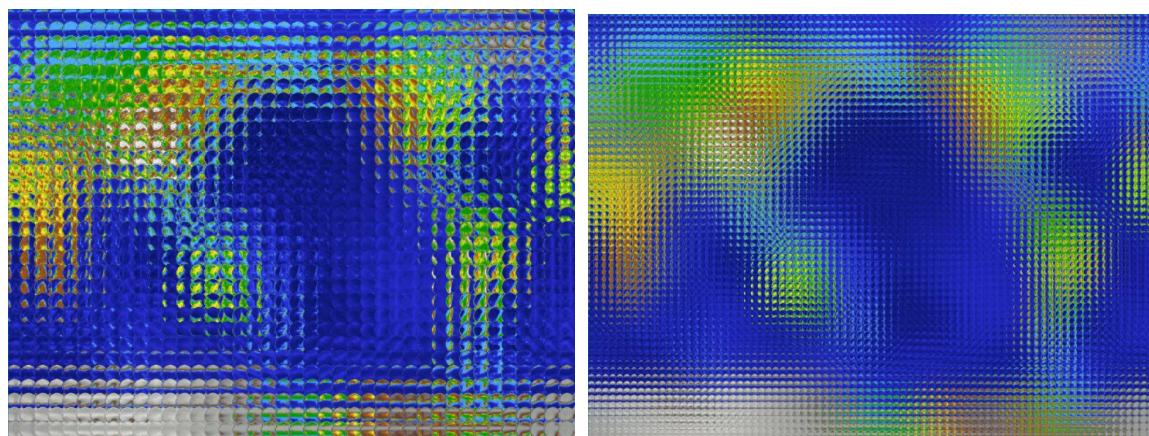
TxNew += In.Tex;

t0 = tex2D(smpDif, TxNew);

Out = t0;
return Out;
}

```

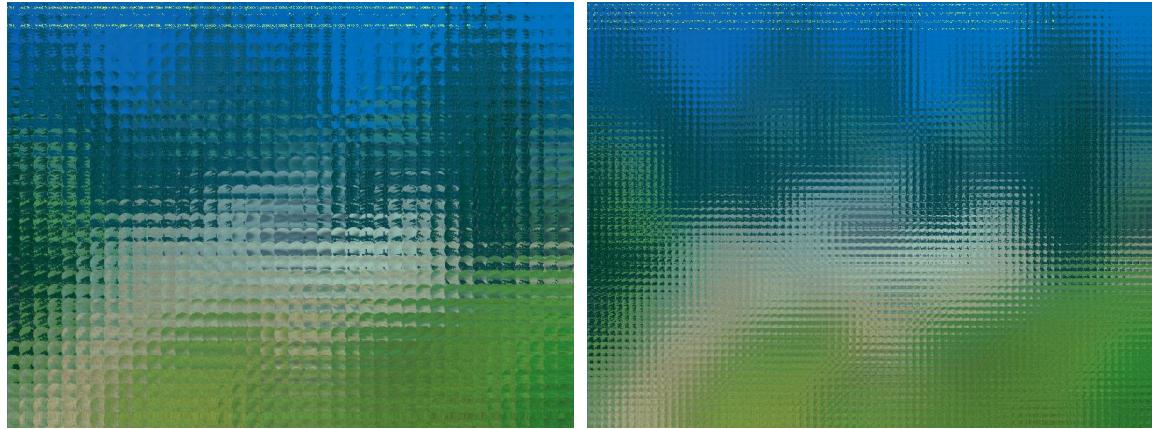
실행하면 다음과 같이 화면에 출력되는데 오른쪽 그림은 좀 더 촘촘히 Embossing텍스처를 만든 것입니다.



<Embossing 텍스처를 적용한 화면: [h4\\_02\\_emboss0.zip](#)>

void CShaderEx::Render() 함수에서 #if 0을 #if 1로 하고 컴파일 한 다음 실행하면 <Embossing 텍스처>가 화면에 출력 됩니다.

이것을 지형에 적용시켜 보면 다음과 같이 출력됩니다.



< Embossing 텍스처를 적용한 화면: [h4\\_02\\_emboss0\\_a.zip](#)

### 6.2.2 무늬 유리 효과 2

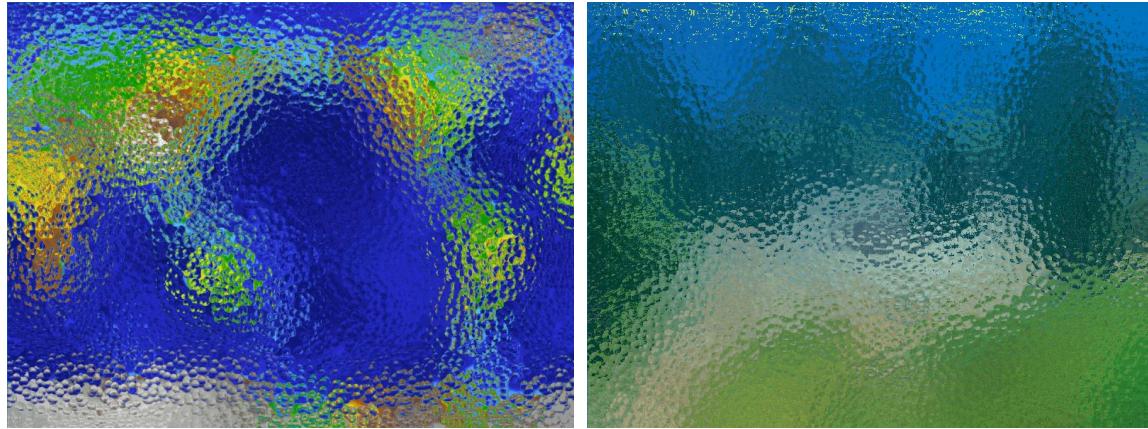
무늬 유리 효과를 만들었는데 같은 간격으로 일정하게 나타나는 것이 좀 단조로운 느낌이 듭니다. 이 번에는 circle 이미지의 위치를 Random하게 만들어서 Embossing 텍스처를 만들어 봅시다. 수정할 부분은 for 문 내용으로 다음과 같이 위치를 rand()함수로 정합니다.

```
for(int i=0; i<20000; ++i)
{
    pVtx[0].p.x = rand()%900 *1.4f - m_fW;
    pVtx[0].p.y = rand()%700 *1.4f - m_fW;

    pVtx[1].p.x = pVtx[0].p.x + m_fW;
    pVtx[1].p.y = pVtx[0].p.y + 0 ;
    pVtx[2].p.x = pVtx[0].p.x + m_fW;
    pVtx[2].p.y = pVtx[0].p.y + m_fW;
    pVtx[3].p.x = pVtx[0].p.x + 0 ;
    pVtx[3].p.y = pVtx[0].p.y + m_fW;

    m_pDev->SetTexture(0, m_pTex1);
    m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(T));
}
```

20000 이란 숫자는 큰 의미는 없습니다. 겹치는 부분이 있을 것 같아 적당히 큰 값을 선택했습니다. 이것을 실행하면 다음과 같습니다.



<Random() 함수를 사용한 무늬 유리 효과: [h4\\_02\\_emboss1.zip](#), [h4\\_02\\_emboss1\\_a.zip](#)>

이전의 같은 간격으로 구성한 것보다 많이 좋아 졌음을 볼 수 있습니다. 여기에 작은 무늬의 종류를 이전에 사용했던 패턴들을 섞어봅시다.

이전 코드와 달라진 것은 작은 무늬를 더 추가한 것과

```
if(FAILED(D3DXCreateTextureFromFile(m_pDev, "Texture/pattern_circle.png", &m_pTex1)))
    return -1;

if(FAILED(D3DXCreateTextureFromFile(m_pDev, "Texture/pattern_ginkgo.png", &m_pTex2)))
    return -1;

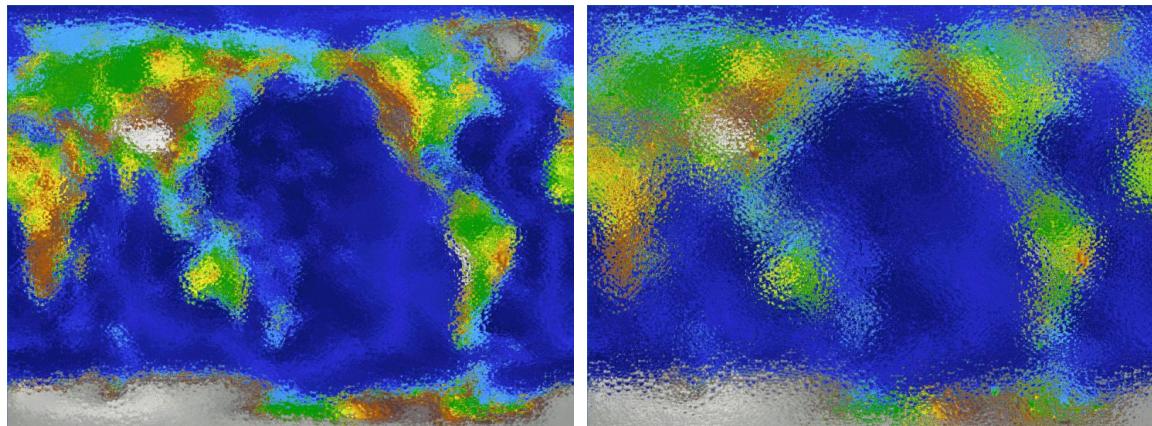
if(FAILED(D3DXCreateTextureFromFile(m_pDev, "Texture/pattern_cube1.png", &m_pTex3)))
    return -1;
```

무늬 유리창을 만드는 For문에서 rand() 함수를 사용해 패턴을 선택하는 것 입니다.

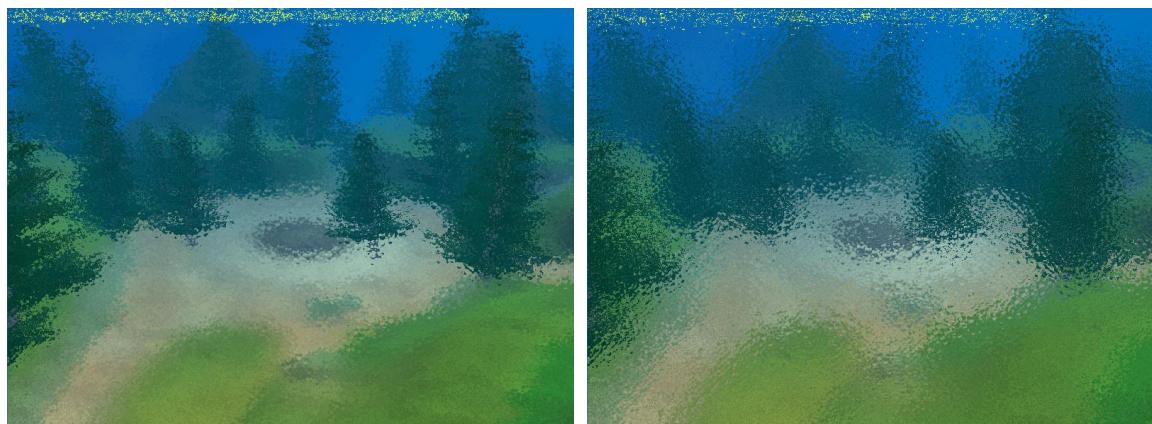
```
for(int i=0; i<20000; ++i)
{
...
    int c = rand()%3;

    if(0 ==c)      m_pDev->SetTexture(0, m_pTex1);
    else if(1 ==c) m_pDev->SetTexture(0, m_pTex2);
    else if(2 ==c) m_pDev->SetTexture(0, m_pTex3);
```

```
m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(T));
}
```



<무늬 유리창 효과: [h4\\_02\\_emboss1.zip](#)>

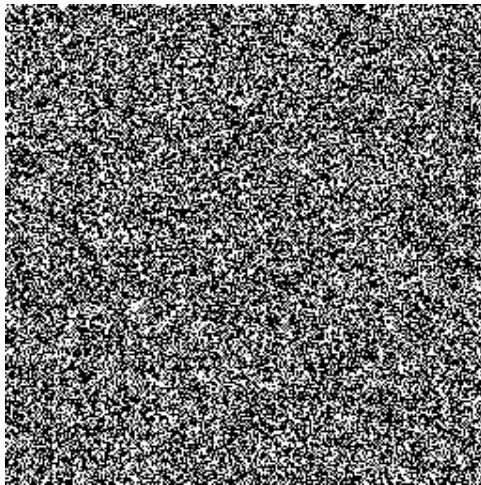


<무늬 유리창 효과: [h4\\_02\\_emboss1\\_a.zip](#)>

하나의 패턴을 사용해서 만드는 것보다 여러 개를 혼합해서 사용하는 것이 더 효과적이라는 것을 알 수 있습니다.

### 6.2.3 화면 잡음(Noise)

아날로그 TV의 경우 전파 신호가 약하면 잡음(Noise)의 효과가 강해져 화면이 좌우로 일그러지는 현상을 볼 수 있습니다. 이런 종류의 Noise를 White Noise 라고도 합니다. 이 효과도 가만히 생각해보면 정상적인 3D 화면 픽셀을 좌우로 이동시키는 것과 유사합니다. 그런데 이 이동 값은 일정한 것이 아니라 거의 Random 한 것입니다. (잡음 또한 Random입니다.)



&lt;노이즈 맵&gt;

만약 Random을 표현할 수 있는 텍스처가 있다면 우리는 바로 앞에서 해왔던 일들을 거의 그대로 적용할 수 있습니다. 간단히 정리하면 Random 이미지에서 픽셀을 가지고 와서 이 값을 정상 화면의 UV를 왜곡시키는 요소(변수)로 만들면 영상 잡음을 만들어 낼 수 있을 것입니다.

Random 이미지는 Adobe의 Photoshop 과 같은 그래픽 툴에서 Noise를 통해서 다음과 같이 만들 수 있습니다. 이것을 노이즈 맵(Noise Map)이라 부르기도 합니다.

이 노이즈 맵의 색상(R, G, B) 중에 R, G를 화면 왜곡 비율로 설정하면 되는데 색상은 [0, 1] 범위이고 U, V도 [0, 1] 범위이므로 색상의 R, G값을 그대로 사용하지 않고 적당히 줄입니다.

이 값을 원본 U, V에 더하면 정상 화면을 왜곡 시키는 좌표가 될 것입니다.

다음은 쉐이더 코드입니다.

```

float m_fRepeatX = 10;           // Repeat for X
float m_fRepeatY = 10;           // Repeat for Y

float4 Px1Proc0(SvsOut In) : COLOR0
{
    float4 Out=0;
    float4 t0= 0;
    float2 t= 0;

    t = In.Tex;
    t.x *= m_fRepeatX;
    t.y *= m_fRepeatY;
    t0= tex2D(smp1, t)-0.5;      // Sampling from Noise Texture ==> [-0.5, 0.5];
    t0 *=0.1f;                   // [-0.05, 0.05];
    t = In.Tex + float2(t0.x, t0.y); // Modified Texture Coordinate
    t0= tex2D(smp0, t);          // Sampling from Diffuse Texture

    Out      = t0;
    return Out;
}

```

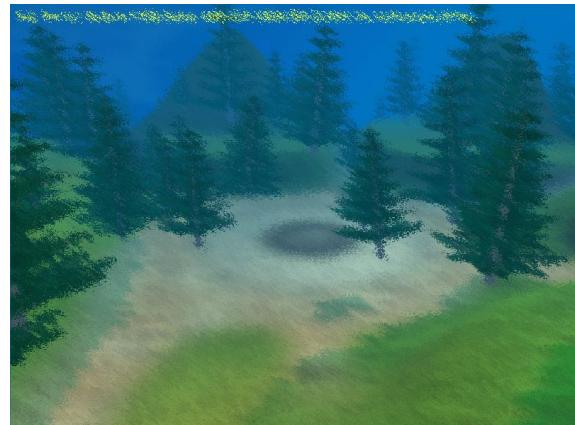
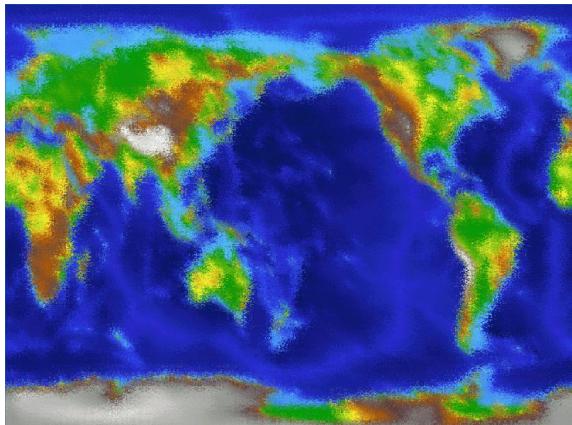
}

코드를 보면 정상화면의 U V에 노이즈 맵의 반복 정도를 X와 Y에 다르게 곱하고 있습니다. 이것은 화면의 가로 세로 방향에 대한 노이즈를 다른 비율로 만들기 위해서입니다.

이 곱한 값을 가지고 노이즈 맵(smp1)에서 색상을 가져온 다음 0.5를 빼서 노이즈가 좌우 또는 상하로 발생하도록 합니다. 그 다음 0.1를 곱해서 적당히 왜곡이 너무 크지 않도록 줄입니다. 이 왜곡 값과 원본 U, V값을 더해 최종 텍스처 좌표로 만든 다음 화면을 저장한 텍스처에서 샘플링 합니다.

반복 횟수 값을 다음과 같이 설정하면 화면에서 잡음의 정도를 볼 수 있습니다.

```
float time = D3DXToRadian( GetTickCount() *0.05f);
float fSin=sinf(time);
float fRepeatX = 1 + (1+ fSin)*10;
float fRepeatY = 1 + (1+ fSin)*10;
...
m_pEft->SetFloat("m_fRepeatX", fRepeatX);
m_pEft->SetFloat("m_fRepeatY", fRepeatY);
```



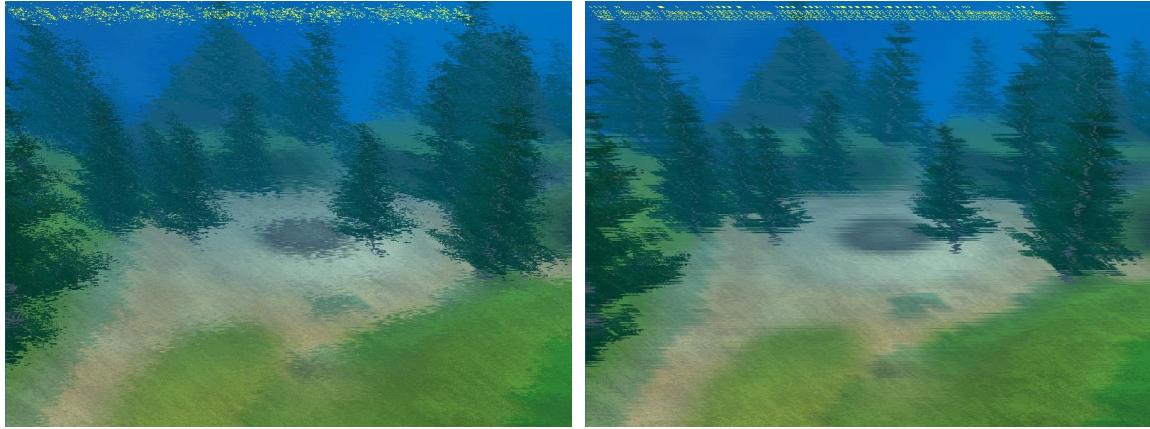
<화면 잡음 효과: [h4\\_02\\_noise.zip](#), [h4\\_02\\_noise\\_a.zip](#)

만약 반복 값 다음과 같이 Random을 적용하면 상하 좌우로 심하게 흔들리는 화면을 얻을 수 있습니다.

```
fRepeatX = (10 + (rand()%100)) * 0.1f;
fRepeatY = (10 + (rand()%100)) * 0.1f;
```

그런데 적당히 좌우로만 잡음이 생기도록 하는 것이 좋으므로 Y방향은 고정하고 X 방향만 Random을 적용합니다.

```
fRepeatX = (1 + rand()%51)/20;
fRepeatY = 20; // Y는 적당한 값으로 고정
```



<화면 잡음 효과: [h4\\_02\\_noise\\_a.zip](#)>

### 6.3 연필 선 효과(Pencil Stroke)

화면을 연필로 스케치한 것처럼 표현하는 방법은 여러 가지가 있습니다. 첫 번째 방법은 3D 모델을 렌더링 할 때 반사의 밝기에 따라 스케치용 텍스처를 매핑 하는 방법이 있고, 두 번째 방법은 색상을 R, G, B로 분리해서 각각의 채도, 명도에 적용하는 방법, 그리고 세 번째 방법은 전체 장면을 각 픽셀의 밝기에 따라 레벨을 정한 텍스처를 혼합하는 방법이 있습니다.

이 장에서는 세 번째 방법인 장면의 밝기에 연필 스케치 이미지를 적용하는 방법을 보이겠습니다.



<연필선 이미지 A>

<연필선 이미지 B>

<연필선 이미지 C>

세 번째 방법을 적용하기 위해서 그림처럼 연필선 이미지 A, B, C를 준비합니다. 보면 알다시피 A 가 가장 밝고 그 다음 B, 마지막은 C인 것을 알 수 있습니다.

이 이미지를 장면에 적용하기 위해서 연필선 이미지 A와 연필 선 이미지 B만 혼합하는 예를 먼저 들어 봅시다.

만약 연필선 이미지 A에 60%, 연필선 이미지 B에 40%를 혼합할 때 어떻게 하면 될까요? 당연히  $A * 0.6 + B * 0.4$  하면 됩니다. 이 수식은 선형 보간과 다름이 없습니다. 따라서 최종 색상은  $= A * 0.6 + (1 - 0.6) * B$  한 것과 다름이 없습니다.

그런데 이렇게 쉬운 내용을 쉐이더 프로그램으로 표현할 줄 알아야 합니다.

의사 코드로 대충 만들어 보면 다음과 같습니다.

```
float W = 0.6;
float4 tA = Sampling(Texture A);
float4 tB = Sampling(Texture B);

float4 Out = tA * W + (1 - W) * tB;
```

이것을 연습장에 작성하신 분은 이미 나머지 내용도 쉽게 만들어 갈 수 있을 것입니다.

만약 W를 장면 텍셀의 밝기로 한다면 앞의 의사 코드는 쉐이더를 이용해서 우리가 연필 선으로 표현하고자 하는 코드와 동일하게 됩니다.

남아 있는 것은 텍셀의 밝기입니다. 이것은 간단하게 흑백 이미지로 만들어서 이 크기를 텍셀의 밝기로 다음과 같이 정하면 됩니다.

텍셀의 밝기  $W = (\text{픽셀 Red}) * 0.301 + (\text{픽셀 Green}) * 0.587 + (\text{픽셀 Blue}) * 0.114$

이제 이것을 쉐이더로 표현하면 다음과 같습니다.

```
sampler smp0 : register(s0);
sampler smp1 : register(s1);
sampler smp2 : register(s2);

float4 Px1Proc0(float2 uv : TEXCOORD0) : COLOR0
{
    float2 tx= uv * 8.0f;           // U,V 좌표를 8배 한다.

    float4 t0 = tex2D(smp0, uv);
    float4 t1 = tex2D(smp1, tx);
    float4 t2 = tex2D(smp2, tx);
```

```

float4 Out =0;
float fMono = t0.r * 0.299 + t0.g * 0.587 + t0.b * 0.114;

float fW = fMono;
Out = fW * t1 + (1-fW) *t2;

return Out;
}

```

쉐이더 코드에서 연필선 효과가 잘 안보여 UV좌표를 8배 했습니다. 또한 쉐이더 코드에서 샘플러 register를 지정해서 사용하고 있어서 프로그램에서는 Address Mode와 필터링을 지정해야 하고 디바이스의 SetTexture() 함수로 register에 텍스처를 전달해야 합니다.

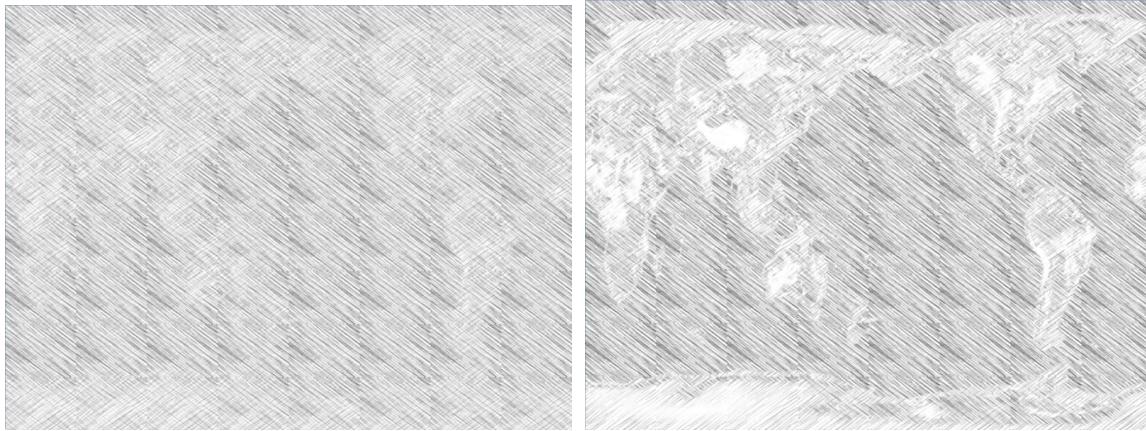
```

for(int i=0; i<4; ++i)
{
    m_pDev->SetSamplerState(i, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP);
    ...
    m_pDev->SetSamplerState(i, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
    ...
}

m_pEft->SetTechnique("Tech");
...
m_pDev->SetTexture(0, m_pTx0);
m_pDev->SetTexture(1, m_pTx1);
m_pDev->SetTexture(2, m_pTx2);
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, m_pVtx, sizeof(VtxDUV1));
...

```

실행파일에서는 이미지를 로드하고 쉐이더로 연결한 것 밖에 없습니다. 이것을 실행하면 다음의 왼쪽과 같은 어렵잖은 윤곽이 잡힌 장면을 볼 수 있습니다.

<연필 선 효과: [h4\\_03\\_hatch1.zip](#)>

우리가 원하는 것은 최소한 오른쪽 장면입니다. 이렇게 되기 위해서는 그림이 3장이 필요합니다. 그런데 가장 밝은 것을 흰색 = (1.0,1.0,1.0,1.0)으로 한다면 그림 추가 없이 쉐이더 코드 만으로도 해결이 가능합니다.

그림 추가는 해결이 되었다고 보고 이제는 3장의 이미지에 대한 레벨을 만들어야 합니다.

간단하게 밝기의 레벨을 3, 2, 1로 정합시다. 그리고 흑백으로 만든 fW값에 3.5를 곱하고 이 값을 비교해 가면서 다음과 같이 각 레벨마다 이웃한 레벨끼리 선형 보간을 합니다.

```

float m_Level0 = 3.0f;
float m_Level1 = 2.0f;
float m_Level2 = 1.0f;
...
float4 White = {1,1,1,1};

float Bright = t0.r * 0.299f + t0.g * 0.587f + t0.b * 0.114f;

Bright *= 3.5f;

if( Bright>=m_Level0)
    Out = White;

else if( Bright>m_Level1)
    Out = (Bright - 2) * White + (3 - Bright)*t1;

else if( Bright>m_Level2)
    Out = (Bright - 1) * t1 + (2 - Bright)*t2;

```

```

else
    Out = t2;
...

```

이것을 실행하면 앞의 오른쪽 그림과 같은 화면을 얻을 수 있습니다. 그런데 위의 코드는 레벨의 값이 3, 2, 1 같은 간격으로 설정되어 있습니다. 만약 같은 간격이 아니라면 if~else 구문이 다음과 같이 수정되어야 합니다.

```

else if( Bright>m_Level11)
{
    float fW = 1- (m_Level10 - Bright)/(m_Level10-m_Level11);

    Out = fW * White + (1-fW)*t1;
}

else if( Bright>m_Level12)
{
    float fW = 1- (m_Level11 - Bright)/(m_Level11-m_Level12);

    Out = fW * t1 + (1-fW)*t2;
}

```

이렇게 수정이 되고 레벨도 임의로 정해서 잘 동작하면 <연필선 이미지 C> 가 있는 그림을 레벨로 추가해서 앞의 코드처럼 다음과 같이 셰이더를 만들 수 있습니다.

```

static float m_Level10 = 3.4f;
static float m_Level11 = 2.4f;
static float m_Level12 = 1.2f;
static float m_Level13 = 0.2f;

sampler smp0 : register(s0);
sampler smp1 : register(s1);
sampler smp2 : register(s2);
sampler smp3 : register(s3);

float4 Px1Proc0(float2 uv : TEXCOORD0) : COLOR0
{
    float4 Out =0;                                // 출력 값

```

```

static const float4 WHITE = {1,1,1,1}; // 흰색 이미지 대신

float2 tx= uv * 8.f;
float4 t0 = tex2D(smp0, uv);
float4 t1 = tex2D(smp1, tx);
float4 t2 = tex2D(smp2, tx);
float4 t3 = tex2D(smp3, tx);

float fW = 0;
float Bright = t0.r * 0.299f + t0.g * 0.587f + t0.b * 0.114f;

Bright *=5;

if(Bright > m_Level0)
    Out = WHITE;

else if(Bright > m_Level1)
{
    fW = (m_Level0 - Bright)/(m_Level0 - m_Level1);
    Out = (1-fW) * WHITE + fW * t1;
}

else if(Bright > m_Level2)
{
    fW = (m_Level1 - Bright)/(m_Level1 - m_Level2);
    Out = (1-fW) * t1 + fW*t2;
}

else if (Bright > m_Level3)
{
    fW = (m_Level2 - Bright)/(m_Level2 - m_Level3);
    Out = (1-fW) * t2 + fW*t3;
}

else
    Out = t3;

Out = pow(Out, 2.f);
Out *= 1.4f;
return Out;

```

```
}
```

장면이 어두워 셰이더 코드에서 pow() 함수를 사용해 명암 대비(Contrast)를 높였습니다. 추가된 그림 이미지로 인해 샘플러 register s3 사용을 지정했습니다. 프로그램에서는 다음과 같이 텍스처를 연결해야 하는 것은 당연합니다.

```
m_pDev->SetTexture(3, m_pTx3);
```



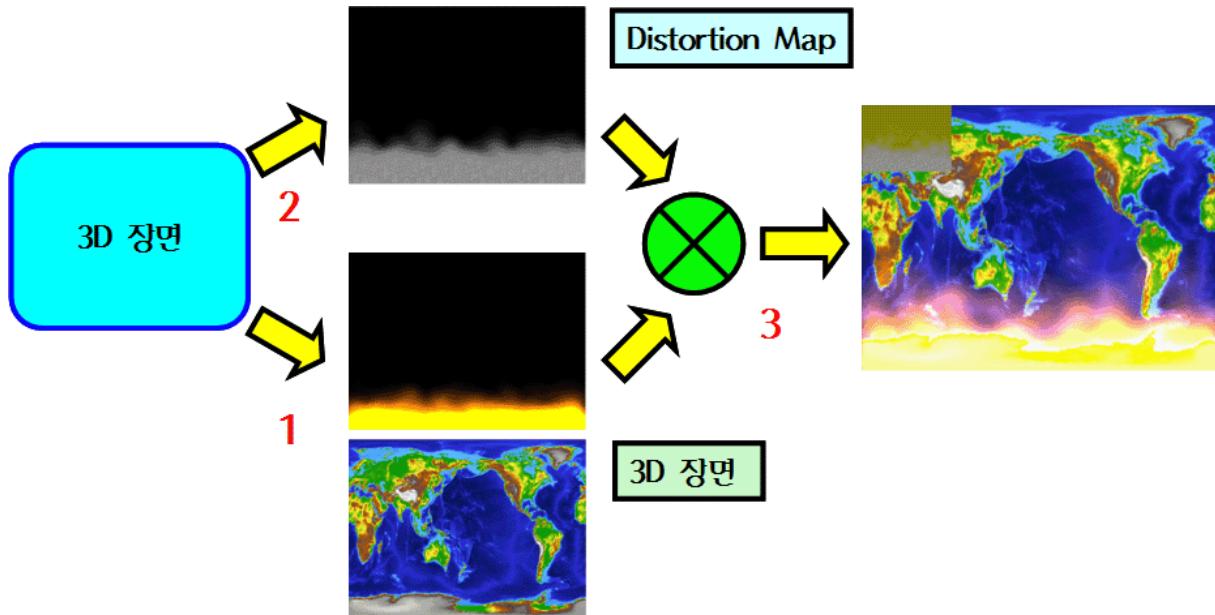
<연필 선 효과: [h4\\_03\\_hatch3.zip](#), [h4\\_03\\_hatch3\\_a.zip](#)>

## 6.4 Distortion (Pencil Stroke)

### 6.4.1 화염 효과(Flame Effect)

Call of Duty - Modern Warfare 같은 게임을 하다 보면 전차의 엔진 뒤의 열기에 의한 아지랑이 효과를 볼 수 있습니다. 또한 폭발 장면에서도 충격파가 등장하고 화염 주변에서도 사실감을 전달하기 위해 파티클(Particle)과 함께 화염 효과를 화면의 왜곡으로 표현된 것을 볼 수 있습니다. 이런 멋진 장면은 게임 프로그래머가 꼭 도전해 보고 싶은 과제이기도 합니다.

이런 충격파, 열기 등 대기 효과들을 화면에 렌더링 하기 위한 구조는 다음 그림과 각 단계로 단순화 시킬 수 있습니다.



<Distortion: 화염 효과 2D>

그림을 통해서 보면 전체 장면은 3부분으로 나누어 렌더링 합니다. 먼저 1 번 단계로 보통 3D를 렌더링 할 때와 마찬가지로 파티클과 장면을 그립니다. 이것을 실시간 텍스처에 저장합니다.

다음으로 화면을 왜곡시킬 파티클을 Distortion 맵으로 사용할 텍스처에 렌더링 합니다. 이 때의 렌더링은 변위 정보가 저장 되어야 합니다. 변위정보의 저장은 파티클에 노이즈 이미지 등을 매핑하고 렌더링 하면 됩니다.

세 번째 단계에서는 3D 장면을 저장한 텍스처와 2번 단계에서 파티클의 변위를 저장한 Distortion 맵을 가지고 혼합합니다. 혼합 방법은 화면 잡음 때와 비슷하게 Distortion 맵의 정보를 가지고 3D 장면을 저장한 픽셀을 상하 또는 좌우로 움직이게 합니다.

방법에 수학적 물리적 이론이 있는 것은 아니니까 여러분은 각 단계 별로 하나씩 해결해 가면 구현의 어려움은 없을 것입니다.

먼저 1단계의 파티클을 화면에 출력해 봅시다. 이 파티클은 [h4\\_04\\_distort0\\_particle.zip](#)에 구현되어 있습니다. class CMcParticle 안에 하나의 파티클에 대한 동작을 제어할 구조체가 다음과 같이 선언되어 있습니다.

```
struct Tpart
{
    D3DXVECTOR2     m_IntP;           // 초기 위치
    ...
    D3DXCOLOR      m_dCol;          // 색상
};
```

- 이 구조체를 가지고 만든 파티클은 CMcParticle::SetPart(int nIdx) 함수 안에 구현되어 있습니다
- 이 구조체와 초기함수로 파티클의 움직임은 CMcParticle::FrameMove() 안에서 채 설정 됩니다.

```

INT CMcParticle::FrameMove()
...
for(i=0; i<m_PrtN; ++i)
{
    CMcParticle::Tpart* pPrt = &m_PrtD[i];

    // 현재 위치 갱신
    pPrt->m_CrnP += pPrt->m_CrnV * ftime;
    float f = (100 - rand()%201) * 0.01f;
    pPrt->m_CrnP.x +=f;

    // 경계 값 설정
    ...
    if(pPrt->m_CrnP.y<0.f || pPrt->m_dCol.a < 0)
        SetPart(i);
}

// 입자의 운동을 Vertex Buffer에 연결
CMcParticle::VtxDRHW* pVtx;
m_pVB->Lock(0,0,(void**)&pVtx, 0);

for(i=0; i<m_PrtN; ++i)
{
    CMcParticle::Tpart* pPrt = &m_PrtD[i];
    ...
}

m_pVB->Unlock();
...

```

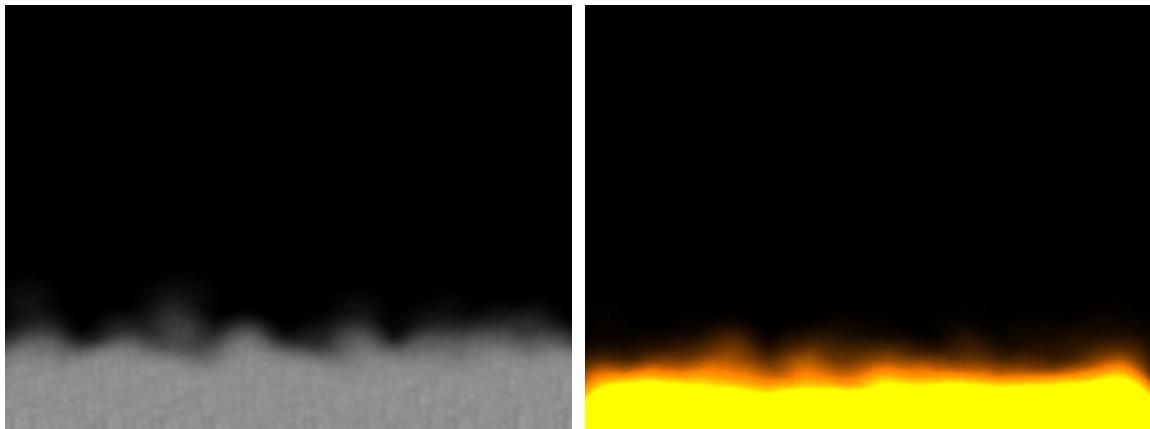
화염이 좌우로 흔들리기 위해 Random 함수로  $f = (100 - \text{rand}() \% 201) * 0.01f;$  값을 만들어서 파티클의 위치에 더했습니다. 파티클을 Point Sprite로 렌더링 하기 위해서 CMcParticle 객체를 만들 때 생성한 정점 버퍼에 위치와 색상을 복사를 합니다.  
파티클은 CMcParticle::Render() 함수에서 렌더링을 합니다.

```

void CMcParticle::Render()
{
...
    m_pDev->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);
    m_pDev->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
    m_pDev->SetRenderState(D3DRS_POINTSIZE, FtoDW(120.f));
    m_pDev->SetRenderState(D3DRS_POINTSIZE_MIN, FtoDW(1.0f));
    m_pDev->SetRenderState(D3DRS_POINTSCALE_A, FtoDW(1.0f));
    m_pDev->SetRenderState(D3DRS_POINTSCALE_B, FtoDW(2.0f));
    m_pDev->SetRenderState(D3DRS_POINTSCALE_C, FtoDW(3.0f));
...
    if(::GetAsyncKeyState('R') & 0X8000)
    {
        m_pDev->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
        m_pDev->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTALPHA);
...
        m_pDev->SetTexture(0, m_pTx);
        m_pDev->SetStreamSource(0, m_pVB, 0, sizeof(CMcParticle::VtxDRHW));
        m_pDev->DrawPrimitive(D3DPT_POINTLIST, 0, m_PrtN);
    }
    else
    {
        m_pDev->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
        m_pDev->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
...
        m_pDev->SetTexture(0, m_pTx);
        m_pDev->SetStreamSource(0, m_pVB, 0, sizeof(CMcParticle::VtxDRHW));
        m_pDev->DrawPrimitive(D3DPT_POINTLIST, 0, m_PrtN);
    }
...
}

```

이 코드를 실행하면 다음 그림의 왼쪽의 화면이 보이며 이 화면을 Distortion 맵으로 만들 것입니다. 'R' 키를 누르면 오른 쪽 그림과 같이 보통 파티클을 렌더링 하는 장면이 나옵니다. 두 장면의 차이는 알파 블렌딩 옵션에서 D3DRS\_DESTBLEND 에 대한 설정 값이 왼쪽그림은 INVSRCALPHA 이고 오른쪽 그림은 DESTALPHA 입니다. 이 점을 꼭 기억하기 바랍니다.



<파티클 효과: [h4\\_04\\_distort0\\_particle.zip](#)>

파티클을 만들었으니 이 파티클을 텍스처에 저장하는 2단계가 남았습니다. 클래스 CTexDistort는 변위 텍스처를 가진 파티클을 화면에 저장하기 위해서 파티클 객체와 IrenderTarget 객체를 가지고 있습니다. IrenderTarget은 장면을 텍스처에 저장하기 위한 객체입니다.

```
class CTexDistort
{
...
    IrenderTarget* m_pTrnd;           // Rendering Target Texture for Scene
    CMcParticle*   m_pPrt;           // Particle Pointer
...
};
```

CTexDistort::FrameMove() 함수는 파티클의 움직임에 대한 정보를 먼저 갱신합니다. 다음으로 이 파티클을 IrenderTarget에 렌더링 합니다. 눈 여겨 볼 것은 색상버퍼를 Clear() 함수로 리셋 할 때 그 값을 D3DXCOLOR(0.5F, 0.5F, 0, 1) 값으로 하고 있습니다. 이 값은 쉐이더 코드에서 x와 y에 대해서 -0.5만큼 이동할 값을입니다. 즉, 색상 정보를 변위로 사용하고 싶은데 색상 값은 항상 양수이므로 쉐이더에서 그 차감만큼 기존 색상에 더해 주어야 하기 때문입니다. 화면을 Clear 하는 것은 또한 특정한 색상으로 채우는 것입니다.

앞서 강조한 알파 블렌딩 상태 D3DRS\_DESTBLEND에 대한 값을 INVSRCALPHA으로 설정함을 주의해야 합니다. 이 것을 바꾸면 변위의 위치가 파티클의 위치와 차이가 생겨 쉐이더 코드에서 더 내용을 추가 해야 합니다.

```
INT CTexDistort::FrameMove()
...
// 파티클 갱신
```

```

m_pPrt->SetTimeEps(g_pApp->m_fElapsedTime);
m_pPrt->FrameMove();

// 파티클을 렌더링 텍스처에 그린다.
m_pTrnd->BeginScene();

m_pDev->Clear(..., D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, D3DXCOLOR(.5F,.5F,0,1), ...);

m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);

m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);
...

m_pDev->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pDev->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
m_pDev->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

m_pPrt->Render();
...
m_pTrnd->EndScene();
...

```

이렇게 파티클을 생성하고 Distortion 맵의 상태를 확인할 필요가 있습니다. 또한 CTexDistort 클래스는 파티클의 화염 효과도 렌더링을 담당하고 있으므로 이 두 가지 일을 동시에 처리하도록 하는 것도 좋습니다. CTexDistort::Render() 함수는 Distortion 맵을 확인하기 위해 이 맵을 화면에 출력하는 것과 장면에서 사용되는 화염 효과에 대한 렌더링 두 가지를 같이 하고 있습니다. Distortion 맵이 확인이 되면 이 부분은 나중에 주석으로 막아야 하는 코드입니다. 다음은 Distortion 맵을 화면에 출력하는 CTexDistort::Render() 함수의 일부 부분입니다.

```

struct Tvtx
{
    FLOAT    p[4];
    FLOAT    u,v;
} pVtx[4] =

```

```

{
    { 0, 0, 0, 1, 0, 0},
    { 200, 0, 0, 1, 1, 0},
    { 200, 150, 0, 1, 1, 1},
    { 0, 150, 0, 1, 0, 1},
};

PDTX pTex = (PDTX)m_pTrnd->GetTexture();

// Distortion 맵을 화면에 출력한다.
m_pDev->SetTexture(0, pTex);
m_pDev->SetFVF(D3DFVF_XYZRHW|D3DFVF_TEX1);
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(Tvtx));
...

```

다음은 CTexDistort::Render() 함수에서 화염 효과를 출력하는 부분입니다.

```

...
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);
m_pDev->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);
m_pDev->SetRenderState(D3DRS_ZENABLE, FALSE);
m_pDev->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);
m_pDev->SetRenderState(D3DRS_LIGHTING, FALSE);
m_pDev->SetRenderState(D3DRS_ALPHATESTENABLE, FALSE);
m_pDev->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
m_pDev->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
m_pDev->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTALPHA);
m_pPrnt->Render();
...

```

지금은 2차원 화면 공간에 RHW로 파티클이 마지막에 그려진다는 전제에 무조건 이전 색상을 덮어쓰기 위해서 ZENABLE를 FALSE로 했습니다. 이것을 3D에서 FALSE로 하면 모든 오브젝트를 통과해서 나타납니다. 3차원 공간에서는 ZWRITEENABLE FALSE하는 것이 더 좋습니다. 자세한 내용은 3D 프

로그램 기초 고정 함수 파이프라인 강좌에 있는 알파 또는 깊이 테스트 부분을 참고 하기 바랍니다.

화염의 색상이 계속 밝아지기 위해서 DESTBLEND, 설정을 D3DBLEND\_DESTALPHA으로 했습니다. 만약 DEST의 알파가 없으면 DEST는 ONE으로 설정하는 것과 동일합니다.

CTexDistort 클래스는 Distortion 텍스처를 쉐이더에서 사용할 수 있도록 텍스처를 가져올 수 있게 다음과 같은 함수도 추가하고 있어야 합니다.

```
PDTX CTexDistort::GetTexture()
{
    return (PDTX)m_pTrnd->GetTexture();
}
```

1, 2 단계의 과정이 다 끝났고, 마지막 단계인 Distortion 맵과 3D 장면을 합치는 3 단계가 남았습니다. 이를 위해 앞에서 계속 쉐이더를 시험하기 위해 사용했던 CShaderEx 클래스가 이 3단계를 처리 할 것입니다.

CShaderEx 클래스는 다음과 같이 CTexDistort 객체를 가지고 있습니다.

```
CTexDistort* m_pDst;
```

CShaderEx 클래스의 Render() 함수는 3 단계를 다음과 같이 처리하고 있습니다. 쉐이더에서 sampler 레지스터를 지정해서 사용하고 있어서 각 다중 텍스처 단계에 대한 주소 모드, 필터링, 등을 전부 설정해야 합니다. 또한 텍스처를 2개를 파이프라인에 걸어주고 있는데 좌표는 1쌍 밖에 없으므로 1 번째 단계의 텍스처를 샘플링 할 때 0번째 텍스처 좌표를 가지고 하도록 다중 텍스처 단계 1 단계를 pDevice->SetTextureStageState(1, D3DTSS\_TEXCOORDINDEX, 0)으로 해야 합니다. 렌더링이 끝나고 코드 마지막에 pDevice->SetTextureStageState(1, D3DTSS\_TEXCOORDINDEX, 1)로 다시 설정해야 하는 것은 당연합니다.

```
void CShaderEx::Render()
...
m_pDev->SetSamplerState(1, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);
m_pDev->SetSamplerState(1, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);
m_pDev->SetSamplerState(1, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
m_pDev->SetSamplerState(1, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
m_pDev->SetSamplerState(1, D3DSAMP_MIPFILTER, D3DTEXF_NONE);
m_pDev->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0);
```

```

FLOAT hHeatHaze = .1f;
PDTX pTex0 = m_pTex;
PDTX pTex1 = m_pDst->GetTexture();

m_pEft->SetTechnique("Tech");
m_pEft->SetFloat("g_HeatHaze", hHeatHaze);
m_pEft->Begin(NULL, 0);
m_pEft->Pass(0);

m_pDev->SetTexture(0, pTex0);
m_pDev->SetTexture(1, pTex1);
m_pDev->SetFVF(VtxDUV1::FVF);
m_pDev->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, m_pVtx, sizeof(VtxDUV1));
m_pEft->End();

m_pDev->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 1);
m_pDev->SetTexture(0, NULL);
m_pDev->SetTexture(1, NULL);
...

```

실행 파일 코드는 다 끝났습니다. 이제 쉐이더 코드가 남아 있습니다. 다음 쉐이더 코드를 보면 이전의 화면 잡음 효과 때와 거의 간단한 쉐이더 코드로 만들어져 있음을 볼 수 있습니다.

```

...
sampler smp0:register(s0);
sampler smp1:register(s1);

SvsOut VtxProc( float3 Pos:POSITION, float2 Tex:TEXCOORD0)
{
    SvsOut Out;
    Out.Pos = float4(Pos,1);
    Out.Tex = Tex;
    return Out;
}

float g_HeatHaze = 0.05;
...

```

```

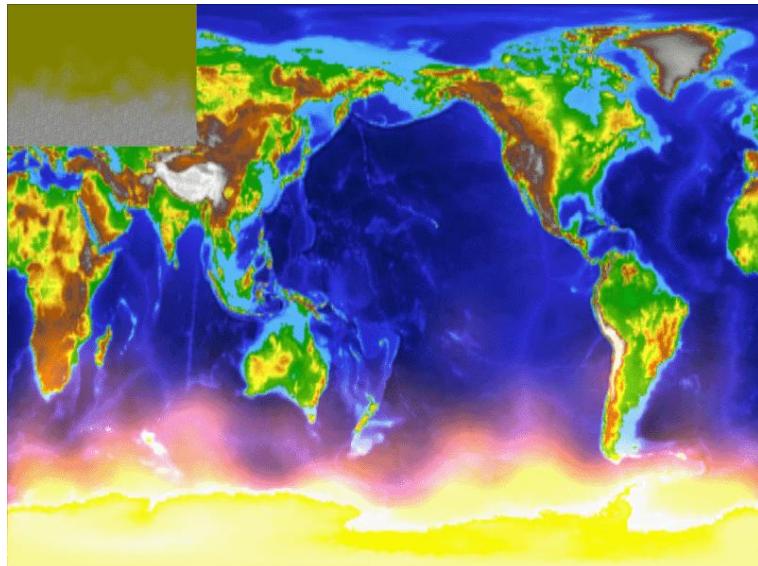
float4 Px1Proc(SvsOut In) : COLORO
{
    float4 Out=0;
    float4 Pert = tex2D(smp1, In.Tex);
    float x = In.Tex.x + (Pert.x-0.5) * g_HeatHaze;
    float y = In.Tex.y + (Pert.y-0.5) * g_HeatHaze;

    Out = tex2D(smp0, float2(x,y));
    return Out;
}

```

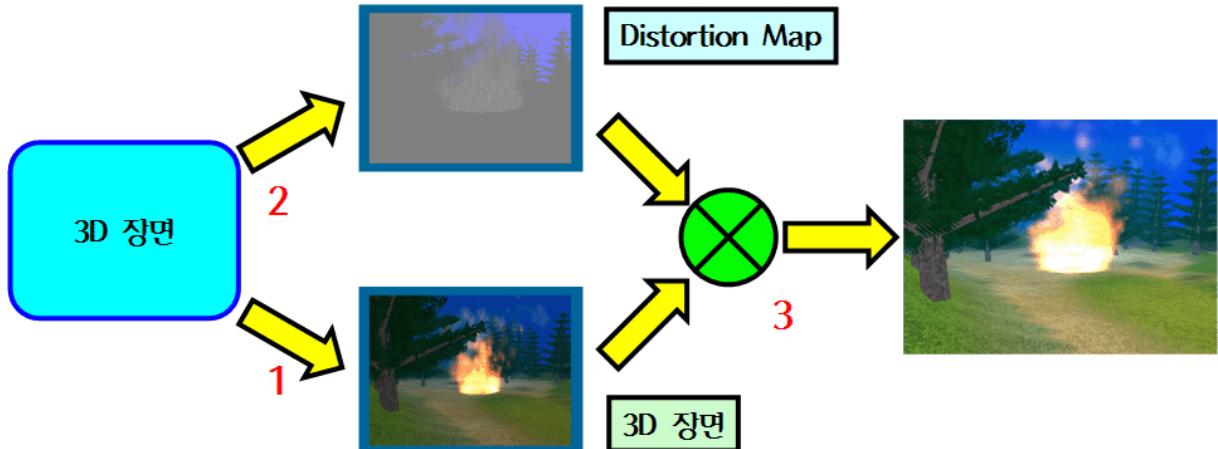
쉐이더 코드의 Pert.x-0.5, Pert.y-0.5 부분 때문에 앞서 Distortion 맵에 파티클을 렌더링 하기 전에 색상 버퍼를 D3DXCOLOR(0.5F, 0.5F, 1, 1)으로 Clear 했습니다.

쉐이더 코드의 픽셀 처리 함수 Px1Proc(SvsOut In)를 보면 1 번째 텍스처의 색상 값을 가져와 0.5씩 빼주고 이 값에 HeatHaze 변수를 곱했습니다. 그리고 다시 원래의 UV좌표에 더했습니다. (Pert.x-0.5) \* g\_HeatHaze 부분이 결국 화면을 왜곡 시키는 요소가 되는 것입니다.



< Distortion 화염 효과: [h4\\_04\\_distort2.zip](#)>

이번에는 이것을 지형이 있는 3D 장면에 만들어 보겠습니다. 먼저 코드를 만들기 전에 여러분은 다음과 같은 그림을 생각하고 있어야 합니다.



<Distortion: 화염 효과 3D>

3D는 그림을 잘 그리면 코드건 수학이건 잘 풀립니다. 따라서 위와 같은 그림을 연습장에 그려놓고 코드 작업을 하는 것이 좋습니다.

이전과 거의 같은데 신경을 써야 하는 부분은 아무래도 Distortion 맵을 만드는 부분입니다. 단순히 파티클만 가지고 이 맵을 만들면 렌더링 오브젝트의 유무에 상관없이 무조건 화면이 혼들릴 것입니다. 따라서 오브젝트 앞에 있으면 효과를 주고 뒤에 있으면 안 주어야 합니다.

간단한 해결 방법은 변위를 만드는 파티클 이외의 나머지 다른 객체들은 이전에 화면을 Clear 할 때 사용했던 값 D3DXCOLOR(0.5F, 0.5F, 1, 1)으로 적용해서 그러면 파티클이 이 오브젝트 뒤에 있으면 Distortion이 적용이 안될 것입니다. 이를 위해 다음과 같은 쉐이더 코드가 필요합니다.

```
float4x4 m_mtWld; // World Matrix;

SvsOut VtxPrcObj( float3 Pos : POSITION, float4 Dif : COLOR0, float2 Tex : TEXCOORD0)
{
    SvsOut Out=(SvsOut)0;
    Out.Pos = mul(float4(Pos,1), m_mtWld);
    Out.Tex = Tex;
    return Out;
}
```

앞의 정점 처리 쉐이더 코드는 정점의 위치를 변화시키고 있고 텍스처 좌표를 픽셀 처리과정으로 넘기고 있습니다. 텍스처 좌표가 필요한 것은 텍스처의 알파 블렌딩 때문입니다. 텍스처의 알파블렌딩이 꺼져 있으면 나뭇잎을 사각형으로 그리게 되어 나무들 사이로 Distortion 효과가 제대로 반영되지 않게 되므로 꼭 텍스처 좌표도 픽셀 처리과정으로 넘겨야 됩니다.

```

float4 Px1PrcObj(SvsOut In) : COLORO
{
    float4 Out={1,1,1,1};

    Out = tex2D(smp0, In.Tex);
    Out.r=0.5f; Out.g=0.5f; Out.b=0.5f;
    return Out;
}

```

출력 색상을 tex2D() 함수로 챔플링 하고 있습니다. 이 것은 출력 색상의 알파 값을 사용하기 위해서입니다. 알파를 사용하기 위해서는 이 오브젝트를 그릴 때 다음과 같이 당연히 알파 블렌딩을 설정해야 합니다.

```

pDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
pDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
pDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );

```

다음으로 R, G, B를 0.5로 설정하고 있습니다. Blue는 Distortion 처리에서 사용하지 않으므로 어떤 값으로 설정해도 상관이 없습니다. 이렇게 해서 Distortion 맵을 만드는 쉐이더 코드는 끝이 났습니다.

이것을 3D 장면에 적용하고 싶은데 이전의 멈춰 있는 화면과 지금의 상황은 많이 다르다는 것을 눈치 챘을 것입니다. 달라진 부분의 첫 번째는 파티클입니다. 이전 파티클은 화면 기준으로 만들어 졌는데 지금 파티클은 3D 공간에서 만들어야 합니다. 여기서는 3D 장면의 파티클 설명은 안 하겠습니다. 일단 파티클이 있다고 가정하고 그것이 이전과 같은 구조로 구성되어 있다면 그냥 같은 방식으로 이용해도 될 것입니다. 추가해야 할 것은 파티클을 3D 장면에 맞게 쉐이더로 다음과 같은 코드로의 수정입니다.

```

SvsOut VtxPrcPtc( float3 Pos : POSITION, float4 Dif : COLORO, float2 Tex : TEXCOORD0 )
{
    SvsOut Out=(SvsOut)0;
    float4 Tpos= mul(float4(Pos,1), m_mtWld);
    Out.Dif = Dif; Out.Tex = Tex;

    Tpos.x *=1.5f; Tpos.y *=1.5f;

```

```

    Out.Pos = Tpos;
    return Out;
}

```

변환이 끝난 후에 마지막에 1.5씩 곱했습니다. 이것은 파티클 크기에만 범위가 적용되면 화면에서는 파티클의 크기가 작으므로 우리가 원하는 효과가 잘 안 나타날 수 있어 좀 더 영역을 키운 것입니다.( 변환이 끝나면 [-1, 1]의 범위가 되므로 1.5란 값은 카메라가 파티클에 가까이 있을 때는 변화가 크게 작용합니다.)

범위를 만드는 파티클의 퍽 셀 쉐이더는 범위 텍스처를 샘플링 한 다음에 파티클 색상의 알파 값을 마지막에 곱해서 사용합니다. 이것은 파티클이 사라지면 범위가 나타나지 않게 하기 위해서입니다. 또한 이런 코드가 제대로 작동하려면 앞서 알파 블렌딩 설정이 Source에는 SrcAlpha가 Dest는 InvSrcAlpha가 설정 되어야 합니다. (이 부분은 파티클 내부에서 처리하는 것이 좋습니다.)

```

float4 PxIPrcPtc(SvsOut In) : COLORO
{
    float4 Out={1,1,1,1};
    float4 t1= In.Dif;

    Out = tex2D(smp0, In.Tex);
    Out.w *= t1.w;
    return Out;
}

```

파티클은 장면을 만들어야 하는 객체이므로 CTexDistort 클래스에서 가져야 할 이유가 없습니다. 그리고 파티클을 통해서 Distortion 맵이 만들어지는 것이라서 파티클 객체가 선언되어 있는 장소에 IrenderTarget 객체를 같이 있는 것이 좋습니다. 이런 이유로 CTexDistort 클래스는 필요 없어지고 나무, 지형이 모여 있는 CMain으로 파티클 객체와 IrenderTarget 객체를 옮겨야 됩니다.

```

class CMain
...
IrenderTarget* m_pTrndS;           // for scene
IrenderTarget* m_pTrndD;           // for distortion

CMcField*      m_pField;
CMcParticle*   m_pPrt ;
...

```

두 번째 달라진 부분은 전체 장면을 텍스처에 Distortion 맵과 장면 텍스처에 두 번 렌더링이 되도록 작성해야 한다는 것입니다. CMain::RenderScene() 함수는 이렇게 2부분으로 거칠게 다음과 같이 코딩하고 있습니다.

```
void CMain::RenderScene(BOOL bDistortion)
{
    ...
    if(FALSE == bDistortion)
    {
        m_SkyBox->Render();
        ...
        m_pField->Render();
        ...
        for(int i=0; i<m_TreeNum; ++i)
        {
            m_pd3dDevice->SetTransform(D3DTS_WORLD, &m_TreeMat[i]);
            m_TreeMsh->Render();
        }
        ...
        m_pPrt->Render(TRUE);
    }
    else {
        ...
        mtVP      = mtViw * mtPrj;
        hr = m_pEft->SetTechnique("Tech");
        hr = m_pEft->Begin(NULL, 0);
        hr = m_pEft->Pass(1);

        mtWld    = mtViw * mtPrj;
        hr = m_pEft->SetMatrix("m_mtWld", &mtWld);
        m_pField->Render();
        ...
        for(int i=0; i<m_TreeNum; ++i)
        {
            mtWld = m_TreeMat[i] * mtVP;
            hr = m_pEft->SetMatrix("m_mtWld", &mtWld);
        }
    }
}
```

```

    m_TreeMsh->Render();
}

...
hr = m_pEft->Pass(2);
mtWld = mtVP;
hr = m_pEft->SetMatrix("m_mtWld", &mtWld);
m_pPrt->Render(FALSE);
hr = m_pEft->End();

...

```

ID3DXEffect에 대한 Pass 도 상당히 많아졌습니다.

```

technique Tech
{
    pass P0           // Distortion Map과 3D 장면 텍스처 합성
    {
        VertexShader = compile vs_1_1 VtxPrcDst();
        PixelShader  = compile ps_2_0 PxlPrcDst();
    }

    pass P1           // Distortion Map에 대한 오브젝트 렌더링
    {
        VertexShader = compile vs_1_1 VtxPrcObj();
        PixelShader  = compile ps_2_0 PxlPrcObj();
    }

    pass P2           // Distortion Map에 대한 파티클 렌더링
    {
        VertexShader = compile vs_1_1 VtxPrcPtc();
        PixelShader  = compile ps_2_0 PxlPrcPtc();
    }
};

```

같은 파일에 있어 CShaderEx와 CMain은 같은 ID3DXEffect 객체를 사용해도 되나 관리의 편의상 각각 따로 ID3DXEffect 객체를 가지고 있도록 했습니다.

CShaderEx 클래스는 Distortion Map의 텍스처 포인터를 설정할 수 있도록 SetDistortionTexture()

함수가 추가되었습니다.

또한 CShaderEx::Render() 함수에서 쉐이더 설정에서 디바이스의 1번 샘플러에게 Distortion 텍스처를 설정하고 있음을 볼 수 있는데 이것이 가능 하려면 쉐이더 코드에서 샘플러를 레지스터에 "sampler '샘플러 객체 이름' : register(s1)"처럼 지정해야 합니다.

...

```
m_pEft->SetTechnique("Tech");
...
m_pDev->SetTexture(1, pTex1);
...
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, m_pVtx, sizeof(VtxDUV1));
```

이렇게 쉐이더 코드와 렌더링 코드를 무사히 마쳤다면 다음과 같은 장면과 같이 화염의 앞에 있는 오브젝트에 대해서는 왜곡이 안 발생하고 불꽃 뒤에 있는 오브젝트에 대해서만 왜곡이 만들어지는 것을 볼 수 있습니다.



< Distortion 3D 장면의 화염 효과: [h4\\_04\\_distort2\\_a\\_1.zip](#)>

ShaderEx.cpp 파일의 #define MYSHADER\_APP 1 설정을 MYSHADER\_APP 0 으로 하면 Distortion 맵이 화면 사이즈보다 축소시켜서 렌더링 하고 있는 것을 볼 수 있습니다.

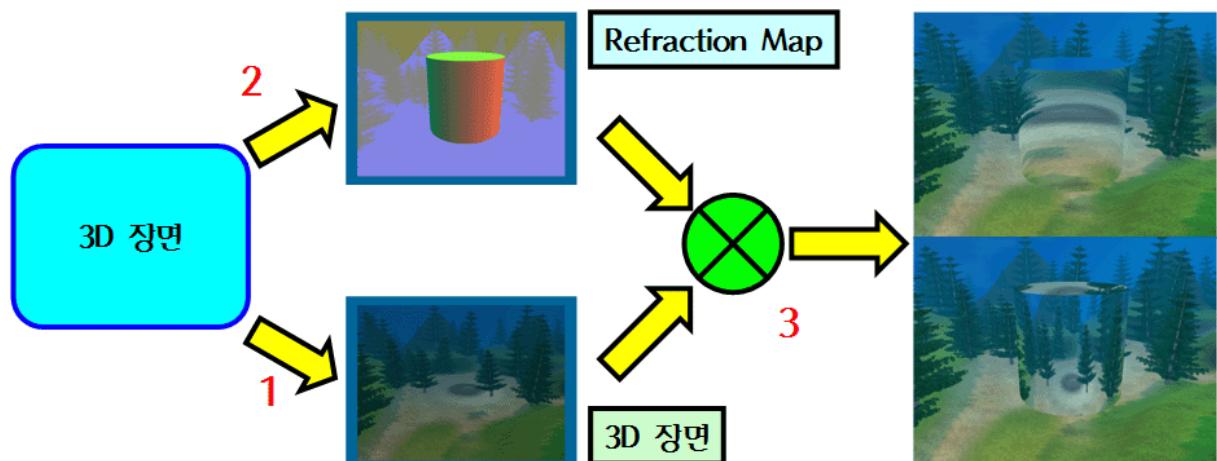
## 6.4.2 굴절 효과(Refraction Effect)

보통 반사나 굴절을 3D에서 표현하려면 환경 맵(Environment Map), 또는 입방체 맵(Cube Map)을 만들어야 합니다. 하지만 앞의 화염 효과를 잘 생각해보면 파티클 대신 어떤 오브젝트에 의해 변위를 만들 수 있지 않을까요?

그러면 오브젝트의 무엇으로 변위를 만들 수 있을까요?

이것도 생각해보면 굴절이라는 것은 투명한 물체의 두께에 의해 발생하는 문제입니다. 그런데 속이 비어있는 3D 렌더링 오브젝트의 두께를 카메라의 시선 방향으로 구하고 굴절을 적용한다는 것은 쉬운 일은 아닙니다.

우리가 만드는 프로그램은 실 세계의 흉내내기입니다. 따라서 적당한 방법을 찾아야 하는데 간단한 방법은 오브젝트의 법선 만큼 픽셀을 이동시키는 것입니다. 이것을 마무리된 코드의 장면을 가지고 그림으로 표현하면 다음과 같습니다.



< Distortion: 3D 장면의 굴절>

앞의 화염 효과 코드에서 2 번의 과정과 이에 관련된 쉐이더 코드를 수정하는 일만 남아있습니다. 먼저 쉐이더 코드입니다.

법선은 픽셀 처리과정에서 사용해야 하므로 쉐이더의 정점 구조체를 다음과 같이 수정해야 합니다.

```
struct SvsOut
{
    float4 Pos : POSITION;
    float4 Dif : COLOR0;
    float2 Tex : TEXCOORD0;
    float3 Nor : TEXCOORD7;
};
```

카메라의 움직임에 의해서 굴절에 대한 오브젝트의 법선은 뷰 변환까지 진행이 되야 합니다.

이를 외부에서 연결할 수 있도록 월드 행렬 \* 뷰 행렬에 대한 변수를 추가합니다.

```
float4x4      m_mtWVP;           // World * View * Projection
float4x4      m_mtWV ;          // World * View
```

굴절 역할을 하는 오브젝트는 법선 벡터를 입력 레지스터에서 받을 수 있도록 float3 Nor:NORMAL0를 추가하고 법선이 카메라의 회전에만 적용되도록 float3x3으로 "월드 \* 뷰 행렬"을 캐스팅해서 변환합니다.

```
SvsOut VtxPrcRfc( float3 Pos : POSITION
                    , float4 Dif : COLOR0
                    , float3 Nor : NORMAL0
                    , float2 Tex : TEXCOORD0)
{
    SvsOut Out=(SvsOut)0;
    float3 N= normalize(mul(Nor, (float3x3)m_mtWV));
    Out.Pos = mul(float4(Pos,1), m_mtWVP);
    Out.Tex = Tex;
    Out.Nor = N;
    return Out;
}
```

파셀 처리과정의 굴절 오브젝트는 텍스처에서 알파 값을, 정점 처리 과정에서 구한 법선 벡터를 색상으로 저장합니다. 그런데 색상은 [0, 1] 범위이고 법선은 [-1, 1] 범위 이므로 법선에 1을 더하고 0.5를 곱해서 저장을 합니다. 출력 색상의 Blue값은 확인용으로 쓰기 위해서 임의 값을 부여했습니다.

```
float4 PxlPrcRfc(SvsOut In) : COLOR0
{
    float4 Out={1,1,1,1};
    float3 Nor= normalize(In.Nor);

    Out = tex2D(smp0, In.Tex);

    Out.r=(Nor.x+1)*0.5;
    Out.g=(Nor.y+1)*0.5;
```

```

    Out.b=0.5f;
    return Out;
}

```

이렇게 색상으로 법선 벡터를 변경해서 저장했습니다. Distortion을 처리하는 쉐이더 함수에서는 이것을 다시 환원하기 위해 굴절 정보를 저장한 텍스처(smp1)에서 색상을 2를 곱한 다음 1을 뺍니다. 이 값을 적당한 굴절 계수를 곱한 후에 다시 전체 장면(smp0)의 픽셀을 샘플링 합니다.

```

static float     g_Dsp = 0.3;

float4 PxlPrcDst(SvsOut In) : COLOR0
{
    float4 Out=0;
    float4 Pert = tex2D(smp1, In.Tex);

    float    x = Pert.x * 2 -1;
    float    y = Pert.y * 2 -1;

    x = In.Tex.x + (x) * g_Dsp;
    y = In.Tex.y + (y) * g_Dsp;

    Out = tex2D(smp0, float2(x,y));
    Out *= 1.5f;
    return Out;
}

```

앞의 VtxPrcRfc(), PxlPrcRfc() 두 함수는 물론 Technique 에 추가해야 합니다.

```

technique Tech
{
...
    pass P3
    {
        VertexShader = compile vs_1_1 VtxPrcRfc();
        PixelShader  = compile ps_2_0 PxlPrcRfc();
    }
};

```

이제 CMain::FrameMove() 함수를 수정할 차례입니다. CMain 클래스에 굴절 효과를 표현할 오브젝트를 추가합니다.

```
ID3DXMesh* m_pCrystall ;
```

이) 객체를 D3DXCreateCylinder() 함수로 생성합니다.

```
D3DXCreateCylinder(m_pd3dDevice, 70, 70, 200, 100, 100, &m_pCrystall, NULL);
```

CMain::RenderScene() 함수에서 이전 파티클에 해당하는 부분을 지우고 굴절 오브젝트를 넣습니다.

...

```
D3DXMatrixScaling(&mtScl, 1, 1, 1);
D3DXMatrixRotationX(&mtRot, D3DXToRadian( 90 ));
D3DXMatrixTranslation(&mtTrn, 300, 40, 250);
```

```
mtWld = mtScl * mtRot * mtTrn;
```

```
mtWV = mtWld * mtView;
```

```
mtWVP = mtWV * mtProj;
```

```
hr = m_pEft->Pass(3);
```

```
hr = m_pEft->SetMatrix("m_mtWVP", &mtWVP);
```

```
hr = m_pEft->SetMatrix("m_mtWV", &mtWV);
```

```
m_pCrystall->DrawSubset(0);
```

CShaderEx::Render() 함수에서 전에 사용한 HeatHaze 변수는 사용을 안 해 지워버립니다.

나머지 코드를 정리하고 실행하면 다음과 같은 화면을 얻을 수 있습니다.



<굴절 효과: [h4\\_04\\_distort3\\_refr.zip](#). g\_Dsp = -0.3, g\_Dsp = 0.3>

그런데 이 예제는 문제가 있습니다. 다음 그림과 같이 카메라가 오브젝트에 가까이 가면 옆으로 퍽셀이 늘어납니다. 이것은 변위를 적용한 U, V가 [0, 1] 범위를 벗어나기 때문입니다. 또한 오브젝트 안쪽으로 들어가면 굴절 효과가 거의 없어집니다. 이런 경우만 제외한다면 완전한 굴절 모양은 아니지만 적당히 사용할 만 합니다.



<굴절 효과 문제>

## 6.5 흐림 효과

### 6.5.1 흐림 효과(Blur Effect) 개요

흐림 효과는 broadening의 일종으로 주변의 퍽셀과 혼합하는 방법입니다. Bloom, Glare 효과 모두 흐림 효과를 기본으로 만든 이펙트입니다. 흐림 효과는 수학으로 표현하면 적분이 됩니다. 참고로 수학의 미분 개념을 이용한 효과는 외곽선 추출, Sharpness가 됩니다.

$(-1, -1)$ g=1	$(0, -1)$ g=1	$(1, -1)$ g=1
$(-1, 0)$ g=1	$(0, 0)$ g=1	$(1, 0)$ g=1
$(-1, 1)$ g=1	$(0, 1)$ g=1	$(1, 1)$ g=1

<3x3 박스 필터(g: 가중치)>

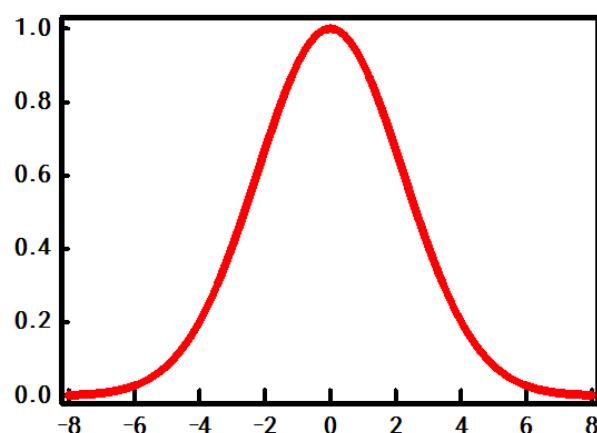
그런데 얼마만큼의 주변 퍽셀과 어떤 방법으로 섞는가? 즉, 혼합에 대한 방법이 많이 있는데 가장 간단한 방법은 옆의 그림처럼 자신과 바로 인접해 있는 9개의 퍽셀에 대한 가중치(Gravitation)를 전부 1로 놓고 이 퍽셀들의 색상에 대한 평균값을 최종 색상으로 정하는 방법입니다.

이 방법은 가장 간단해서 사용된 방법이지만 더 근본적으로 퍽셀 쉐이더 2.0 미만에서 한 번에 샘플링 할 수 있는 텍스처의

수가 많아야 6개 정도이고 이것도 핵셀 쉐이더 버전 1.4를 지원하는 ATI 제품일 때만 가능한 것입니다. 더 하위버전인 1.3 까지는 최대 4개만 샘플링이 가능합니다. 따라서 낮은 버전에 맞추어 허리 필터를 위와 같은 방식으로 만들었던 것입니다.

그런데 요즘은 대부분의 그래픽 카드가 핵셀 쉐이더 2.0이상을 지원합니다. 2.0의 경우 16개의 핵셀을 동시에 샘플링 할 수 있으며 이것은 하나의 텍스처에서도 16번의 샘플링이 가능하다는 것입니다. 이렇게 한 번에 한 텍스처에 16번의 핵셀을 가져와 사용할 수 있는데 앞서 가중치를 동일하게 주고 평균을 내는 것은 개선이 되어야 합니다.

즉, 가운데 있는 핵셀로부터 멀리 떨어질수록 거리에 따라 가중치를 다르게 설정해야 합니다. 이 때 가우스 분포 함수(Gaussian Distribution Function)를 사용합니다.



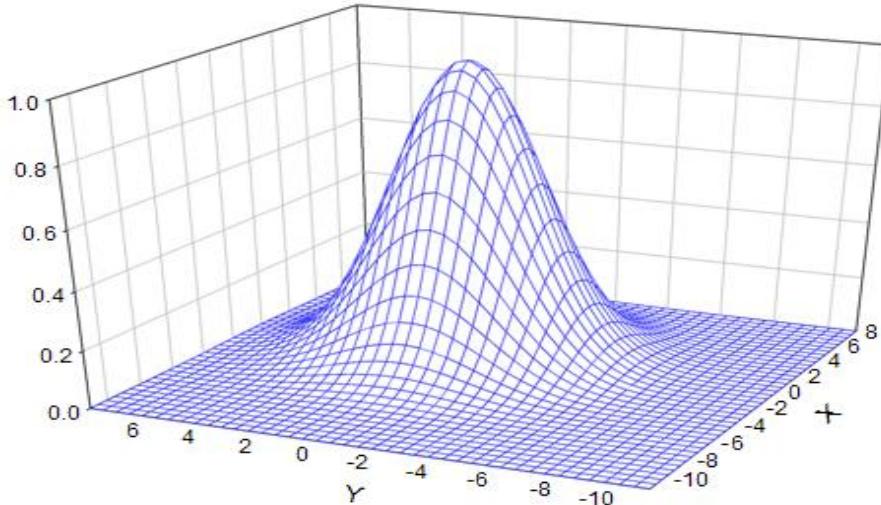
<2차원 가우스 분포함수>

해석을 하는 용도로 두루 사용이 되는 함수입니다.

따라서 컴퓨터 그래픽스에서도 이웃한 핵셀과 혼합할 때 그 가중치를 가우스 분포 함수의 값을 이용합니다.

이 함수를 사용하는 이유는 자연계에서 시간에 따라 확률이 변화하지 않는 경우에 대해서 그 확률에 맞게 어떤 일이 발생하는 일들을 그래프로 표현해 보면 이 가우스 분포 곡선과 일치하게 됩니다. 예를 들어 동전 100개를 가지고 동시에 던졌을 때 앞면이 100개, 99개, 98개, …, 0개가 나오는 빈도를 그래프로 찍어보면 이 가우스 분포 함수와 거의 일치합니다.

가우스 함수는 일반 적으로 Random 한 데이터에서 나타나는 특성으로, 통계뿐만 아니라 물리, 화학, 사회과학 등에서 사건에 대한



&lt;3차원 가우스 분포 함수&gt;

$$\text{가중치(Weight)} = \text{가우스 분포함수} = I_0 \exp\left(-\frac{\Delta x^2 + \Delta y^2}{\Delta^2}\right)$$

$I_0$  : Intensity(상수),  $\Delta x$  : 중심 픽셀과 x 방향으로의 거리,  $\Delta y$  : 중심 픽셀과 y 방향으로의 거리,  $\Delta^2$  : 분산 값

만약 중심 픽셀의 가중치를 1로 놓게 되면  $\Delta x=0$ ,  $\Delta y=0$  이 되어  $I_0=1$  이 되어야 합니다.

게임프로그래밍에서는 대부분  $I_0=1$ 로 설정하고  $\Delta^2$  만 조절할 수 있도록 다음과 같이 간략하게 만들어 사용합니다.

$$\text{가중치} = \exp(-(\Delta x^2 + \Delta y^2) * \Delta)$$

쉐이더 프로그램은 픽셀 샘플링의 제한이 있어 이렇게 X, Y 방향으로 동시에 처리하지 않고 한 방향씩 처리합니다. 따라서 가중치 계산은 1차원에서만 계산이 되며 다음은 가장 많이 사용이 되는 형태입니다.

$$\text{가중치}(W) = \exp(\Delta r^2 * \Delta)$$

이 함수에 익숙해지기 위해 예를 들어 봅시다. 만약  $\Delta=-0.08$  일 때 거리에 따라 가중치를 구하면 다음과 같습니다.

$$\Delta r=0: 0^2 = 0. \quad \text{가중치} = \exp(0 * (-0.08)) = 1$$

$\Delta r = 1: 1^2 = 1$ . 가중치 =  $\exp(-0.08) = 0.9231$   
 $\Delta r = 2: 2^2 = 4$ . 가중치 =  $\exp(-0.08 \times 4) = 0.7261$   
 $\Delta r = 3: 3^2 = 9$ . 가중치 =  $\exp(-0.08 \times 9) = 0.4868$   
 $\Delta r = 4: 4^2 = 16$ . 가중치 =  $\exp(-0.08 \times 16) = 0.2780$   
 $\Delta r = 5: 5^2 = 25$ . 가중치 =  $\exp(-0.08 \times 25) = 0.1353$   
 $\Delta r = 6: 6^2 = 36$ . 가중치 =  $\exp(-0.08 \times 36) = 0.0561$   
 $\Delta r = 7: 7^2 = 49$ . 가중치 =  $\exp(-0.08 \times 49) = 0.0198$   
 가중치의 합 = 6.2108

가중치의 합은 범위가 -7, -6, -5, ..., 5, 6, 7에 해당하는 가중치에 대한 총합입니다.  
가중치를 구했으면 최종 색상은 다음과 같이 결정이 됩니다.

$$\text{최종 색상} = \frac{1}{\text{Total Weight}} \sum W_i * \text{Pixel}_i$$

이것을 쉐이더에 대한 의사(Pseudo-do) 코드로 작성하면 다음과 같습니다.

```

Weight[N] ← {"사용자 지정"}, Total
"최종 색상" ← 0
for(i←0; i<N; i←i+1)
{
  "새로운 UV" ← "중심 픽셀 UV" + float2( i * "픽셀 가로 폭", 0)
  "최종 색상" ← "최종 색상" + Weight[i] * tex2D(sampler0, "새로운 UV");
}
"최종 색상" ← "최종 색상"/Total;
  
```

여기서 텍스처 좌표는 [0, 1] 이므로 픽셀의 가로 폭 = 1/"텍스처의 가로 폭" 이 됩니다.

이것을 [h4\\_05\\_blur2\\_1.zip](#) 의 "data/hls1.fx" 파일에 픽셀을 처리하는 함수에 다음과 같이 구현되어 있습니다.

```

float m_TexW = 800;
float m_TexH = 600;

static const float Weight[13]=
{
  
```

```

0.0561, 0.1353, 0.278, 0.4868, 0.7261, 0.9231,
1, 0.9231, 0.7261, 0.4868, 0.278, 0.1353, 0.0561};

static const float Total = 6.2108;

float4 Px1BlurX(SvsOut In) : COLOR0
{
    float4 Out=0;

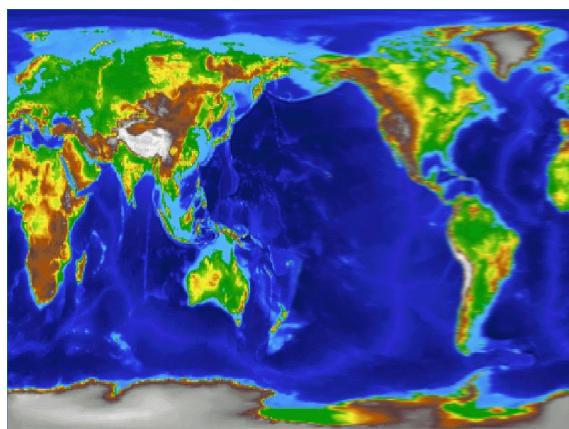
    float2 t = In.Tex;
    float2 uv = 0;
    float tu= 1./m_TexW;

    for(int i=-6; i<6; ++i)
    {
        uv = t+ float2(tu *i, 0);
        Out += Weight[6+i] * tex2D(smp0, uv);
    }

    Out /=Total;
    return Out;
}

```

[h4\\_05\\_blur2\\_1.zip](#)을 실행하면 다음 그림과 같이 오른쪽 화면에 흐림 효과를 적용한 것이 보일 것입니다. 흐림 효과를 저장하기 위해서 CShaderEx 클래스는 IrenderTarget 객체를 하나 가지고 있습니다.



<X 방향 흐림 효과: [h4\\_05\\_blur2\\_1.zip](#)>

X 방향 흐림 효과는 이전의 박스 흐림 필터와 눈에 띄게 큰 차이를 보이지 않습니다. 이 것을 한번 더 Y 방향으로 처리해 봅시다. 쉐이더 코드는 X 방향으로 처리할 때와 거의 같으며 단지 텍스처의 높이와 Y 축으로 샘플링 되도록 u, v를 수정하는 것뿐입니다.

```
float4 Px1BlurY(SvsOut In) : COLOR0
...
    float    tv= 1./m_TexH;

    for(int i=-6; i<6; ++i)
    {
        uv = t+ float2(0, tv *i);
        Out += Weight[6+i] * tex2D(smp0, uv);
    }
...

```

이 함수 또한 Technique에 추가 해야 합니다. 다음으로 Y 방향으로 처리한 결과를 저장하기 위해서 IRenderTarget 객체를 하나 더 추가하고 X 방향으로 흐림 효과를 처리한 텍스처를 Y 방향으로 다시 흐림 효과를 적용합니다. 이 과정은 [h4\\_05\\_blur2.zip](#)의 CShaderEx::FrameMove() 함수에 다음과 같이 구현되어 있습니다.

```
// 원 장면 텍스처
pTx= m_pTex;

// X 방향 흐림 효과를 저장
m_pTrndX->BeginScene();
...
hr = m_pDev->SetTexture(0, pTx);
...
hr = m_pEft->Pass(0);
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, m_pVtx, sizeof(VtxDUV1));
...
m_pTrndX->EndScene();

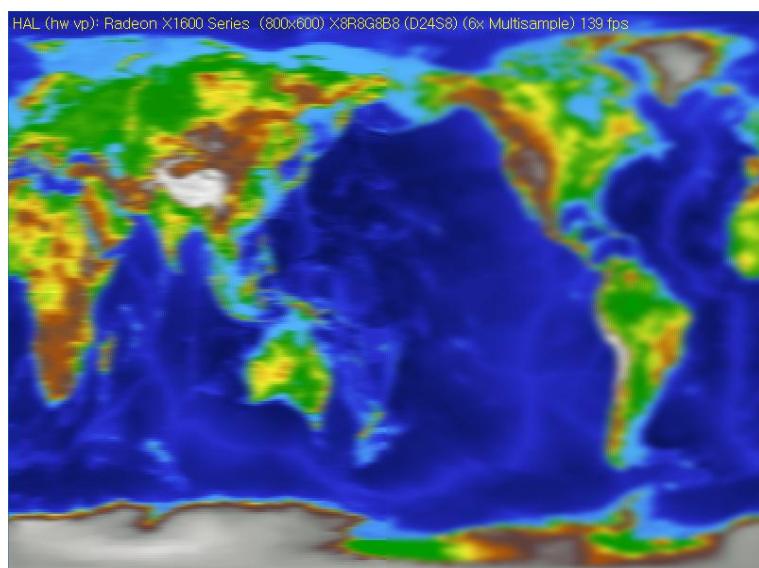
// Y 방향 흐림 효과를 저장
m_pTrndY->BeginScene();
```

```

...
pTx= (PDTX)m_pTrndX->GetTexture();      // X 방향 흐림 효과를 저장한 텍스처를 가져온다.
hr = m_pDev->SetTexture(0, pTx);
...
hr = m_pEft->Pass(1);
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, m_pVtx, sizeof(VtxDUV1));
...
m_pTrndY->EndScene();

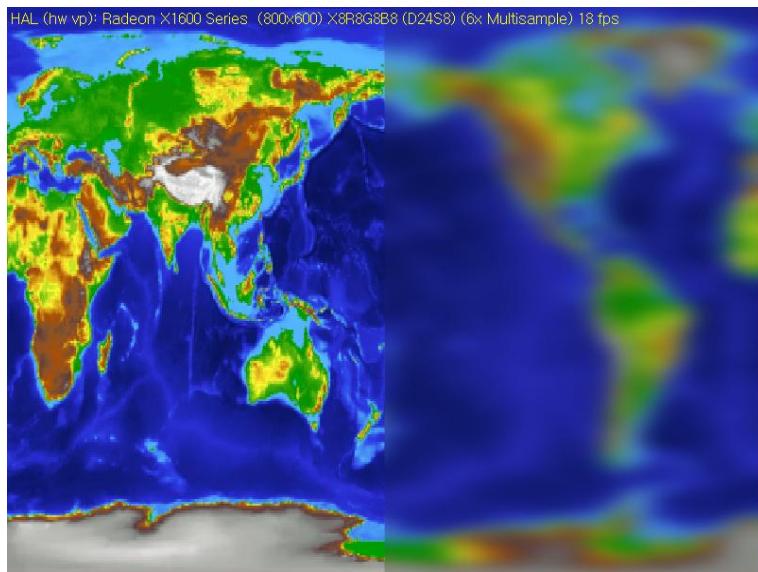
```

다음 그림의 왼쪽 부분은 X 방향으로만 흐림 효과를 적용한 것이고 오른쪽 부분은 왼쪽에서 처리한 것을 Y 방향으로 다시 처리한 결과입니다



<흐림 효과: [h4\\_05\\_blur2\\_2.zip](#) X, Y 양방향으로 적용>

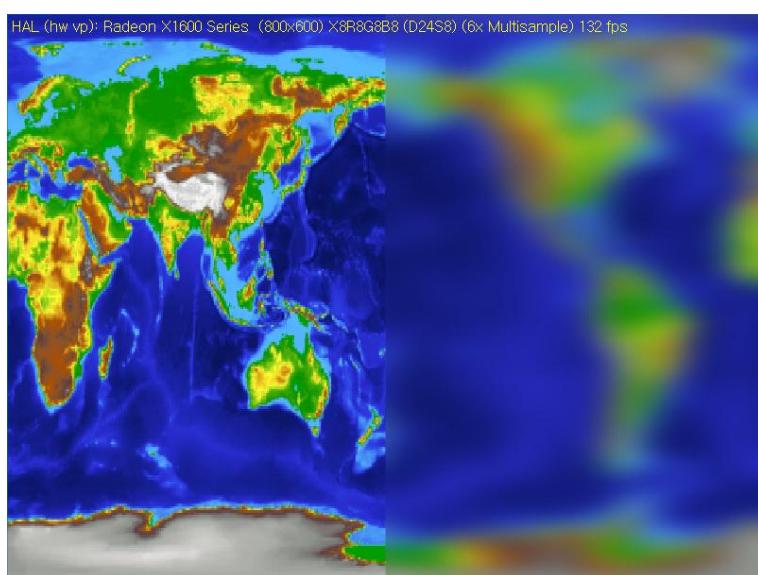
만약 흐림 효과를 화면 전체에 적용한다고 했을 때 X, Y에 대해서 반복적으로 흐림 효과를 적용해야 할 것입니다. 다음은 X, Y 방향에 대해서 각각 8번씩 흐림 효과를 반복적으로 적용한 그림입니다.



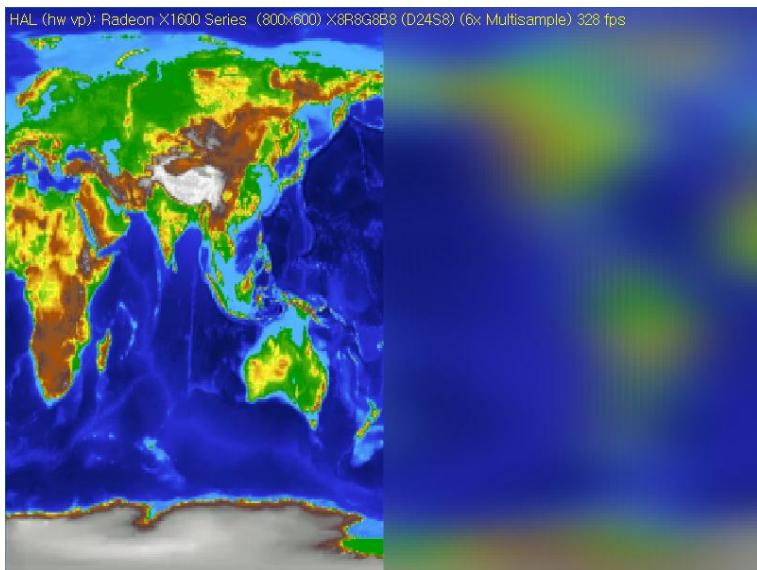
&lt;흐림 효과: h4\_05\_blur2\_2.zip&gt;

예상대로 반복한 만큼 많이 흐려졌습니다. 그러나 문제는 렌더링 속도입니다. 이렇게 렌더링 속도를 깎아 먹으면 다른 효과는 시도도 못할 것입니다. 문제를 해결하기 위해서 흐림 효과라는 것을 되돌아 보면 주변 픽셀과 뭉개기 인데 이 결과를 저장하는 텍스처의 크기를 굳이 화면 크기와 같은 크기로 만들 필요가 없다는 것입니다.

그래서 Blur를 저장할 텍스처의 크기를 줄여서 출력해 보았습니다.

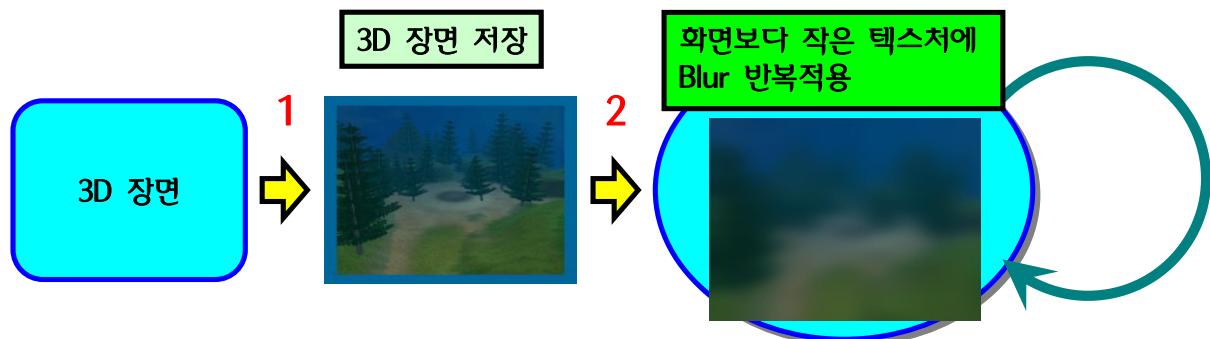


&lt;흐림 효과. Blur 4회 반복. 텍스처 크기: 화면 x (1/2)&gt;

<흐림 효과. 화면 x (1/4). [h4\\_05\\_blur2\\_2.zip](#)>

Blur 횟수를 절반으로 줄이고 텍스처의 크기는 1/2 이하로 줄여도 화면의 효과는 비슷합니다. 대신 렌더링 속도는 이전과 비교할 수 없을 정도 향상됩니다. 즉, 굀셀을 뭉갤 때는 화면 크기보다 작은 텍스처를 가지고 작업하는 것입니다.

이렇게 테스트 코드를 만들었는데 3D 장면에 적용해 봅시다. 3D에 적용되는 과정은 다음 그림과 같을 것입니다.



&lt;흐림 효과 적용 방법&gt;

이전 효과들에서 사용했던 코드에 적용해 봅시다. CShaderEx 클래스는 흐림 효과를 FrameMove() 함수에서 처리하고 있습니다. 따라서 CMain::FrameMove() 함수에서 장면을 텍스처에 저장하고 나서 CShaderEx 객체에 장면에 대한 텍스처 포인터를 전달해야 합니다.

```
HRESULT CMain::FrameMove()
```

```
...
```

```

m_pTrnd->BeginScene();
hr = m_pd3dDevice->Clear(...);
RenderScene();
m_pTrnd->EndScene();

// CShaderEx클래스3D 장면 텍스처 연결
LPDIRECT3DTEXTURE9 pTx = (LPDIRECT3DTEXTURE9)m_pTrnd->GetTexture();
m_pShader->SetSceneTexture(pTx);
SAFE_FRMOV(m_pShader);

```

쉐이더 코드의 sampler에 대한 객체 설정에서 sampler smp0:register(s0); 대신 다음과 같이 수정했습니다.

```

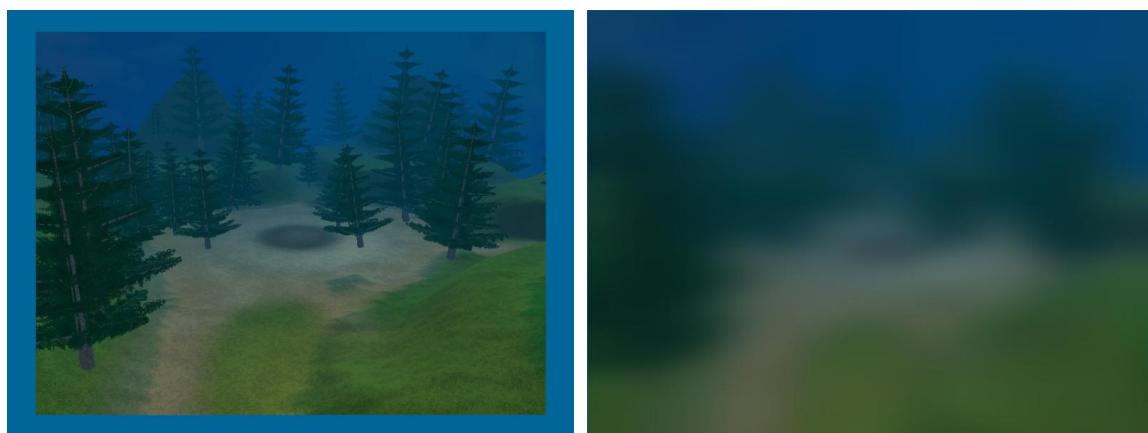
texture m_TxDif;
sampler smpDif = sampler_state
{
    texture = <m_TxDif>;
...
};

```

따라서 디바이스의 SetTexture() 함수가 아닌 이펙트 객체의 SetTexture() 함수를 사용해야 합니다.

```
m_pEft->SetTexture("m_TxDif", pTx);
```

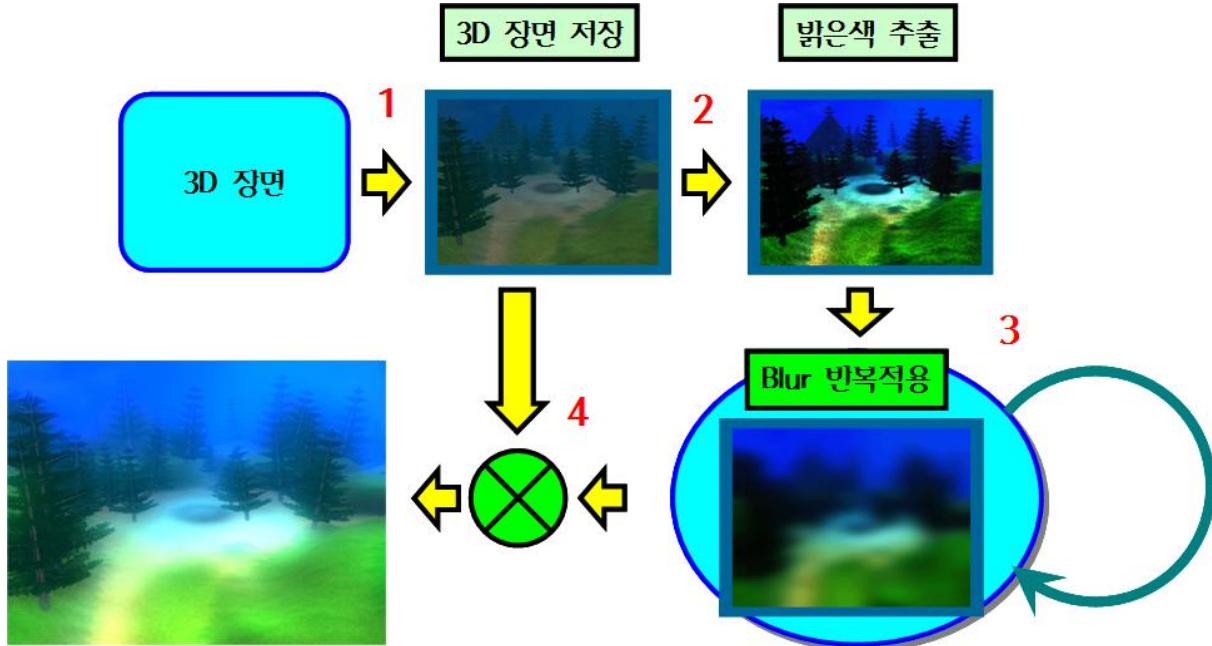
모든 코드는 [h4\\_05\\_blur2\\_a.zip](#)에 있으며 실행하면 다음과 같은 장면을 만들어 냅니다.



<흐림 효과 적용: [h4\\_05\\_blur2\\_a.zip](#)>

### 6.5.2 Glare Effect

흐림 효과는 포스트 이펙트에서 아주 자주 사용되는 기술입니다. 이 방법은 조금만 수정하면 Glare 효과에 쉽게 적용할 수 있습니다. Glare 효과는 아주 밝은 빛 아래의 몽환(夢幻)적 표현이나 어두운 곳에서 갑자기 밝은 곳으로 나왔을 때 등 많은 부분에서 자주 사용됩니다.



<Glare 효과 적용 순서>

지금까지의 내용에 이 효과를 구현하려면 앞의 그림처럼 먼저 장면을 저장한 텍스처에서 밝은 부분을 추려냅니다. 밝은 부분을 추려내는 방법은 쉐이더의 `pow()` 함수를 이용합니다. 이 함수는 조명의 풍 반사에서 설명했듯이 어떤 수에 승수를 해주는 함수입니다. 장면의 픽셀에 이 함수를 사용하면 밝은 부분을 얻어낼 수 있고, 이렇게 얻은 픽셀에 흐림 효과를 적용합니다. 마지막에는 전체 장면과 적절한 연산을 합니다.

2단계의 밝은 색 추출은 다음과 같이 `pow()` 함수를 이용합니다.

```
float4 Px1sharp(SvsOut In) : COLORO
{
    float4 Out = 0;
    float4 t0 = tex2D(smpDiff, In.Tex);
    Out = pow(t0, 4);
    Out *= 30;
    Out.w = 1;
```

```

    return Out;
}

```

다음은 4단계에서 장면 텍스처와 흐림 효과 텍스처를 섞는 쉐이더 코드입니다.

```

float4 Px1Multi(SvsOut In) : COLORO
{
    float4 Out=0;
    float4 t0 = tex2D(smpDif, In.Tex);
    float4 t1 = tex2D(smpDiB, In.Tex);

    //Out = t0 + t1*2.5f;
    Out = t0*.7 + t1*.7f;
    Out *= 1.5f;
    Out.w = 1;
    return Out;
}

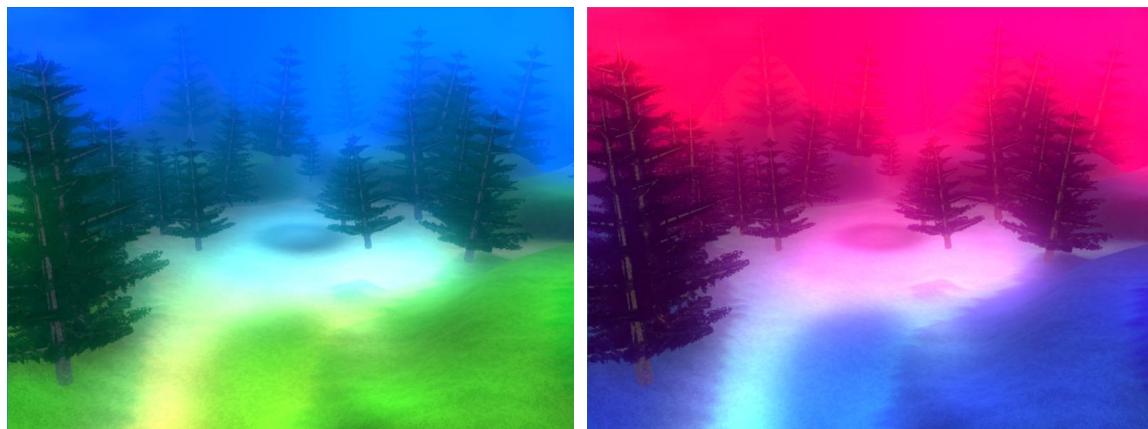
```

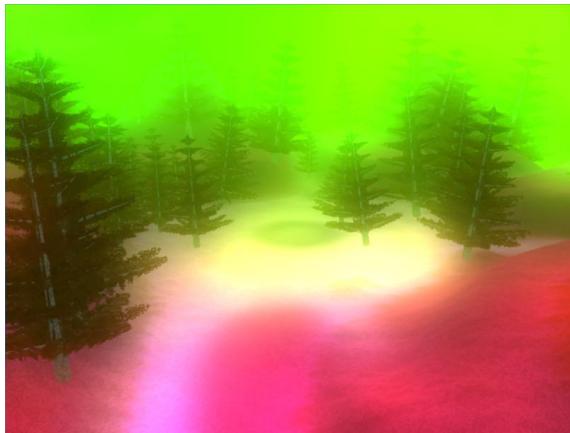
전체적으로 2단계가 더 추가되어 Technique도 수정해야 합니다. [h4\\_05\\_blur3\\_a.zip](#) 예제는 가우스 분포 exp ()함수를 직접 사용하고 있습니다. 이전 결과와 거의 차이가 없게 구성했습니다.

```

for( int i=-fBgn; i<=fBgn; ++i)
{
    uv = In.Tex + float2(i*fInc/m_TexW, 0);
    Out += tex2D(smpDif, uv) * exp( -i*i * fDelta);
}

```



<Glare 효과: [h4\\_05\\_blur3\\_a.zip](#). RGB, BRG, GBR교환>

만약 좀 더 밝게 만들고자 한다면 밝은 부분만 추려내는 쉐이더의 Px1sharp()함수의 내용을 수정해야 합니다.

### 6.5.3 Cross Filter Effect



&lt;DXSDK HDR 예제&gt;

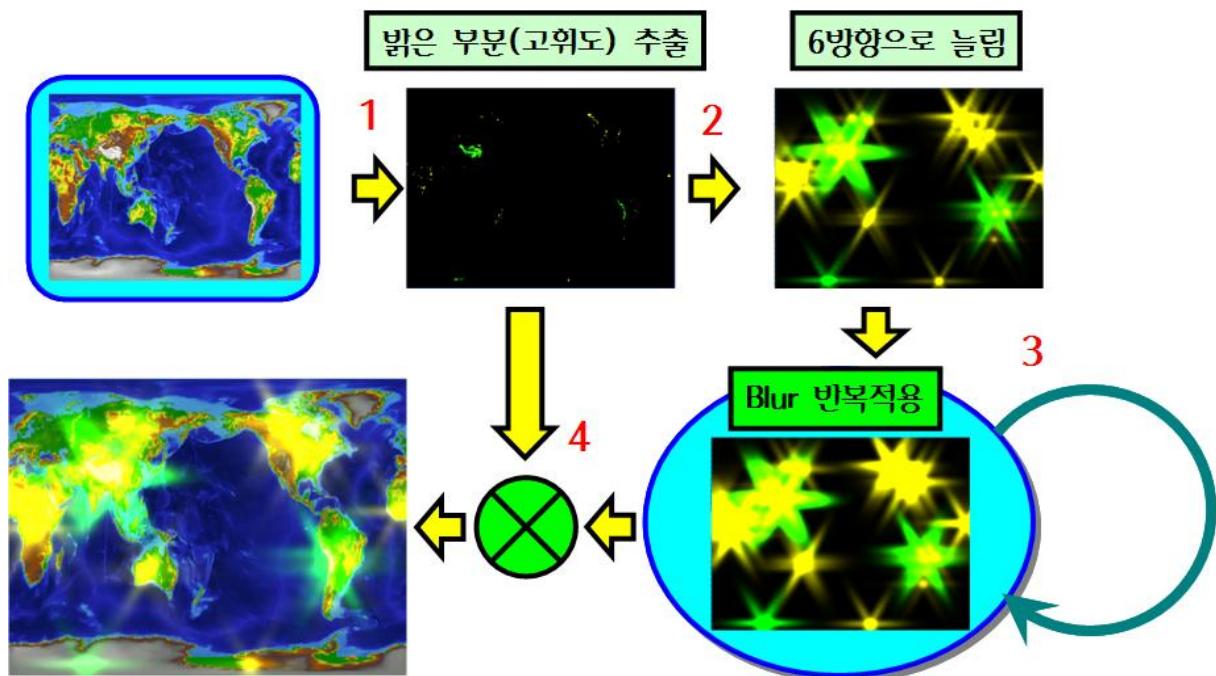
유리나 안경 등에서 강한 빛이 여러 개의 방향으로 분산되는 현상을 가끔 볼 수 있는데 카메라에서는 이런 효과를 극대화하기 위해 크로스 필터(Cross Filter)라는 것을 사용합니다. DXSDK의 HDRLighting 예제는 HDR(High Dynamic Range)에 대한 장면을 연출하고 있습니다.

예제의 설명은 SDK 안에서도 자세히 설명하고 있으니 도움말을 참고하기 바랍니다. 여기서는 이 예제에 사용된 크로스 필터만 만들어 보겠습니다.

크로스 필터를 적용하는 방법은 다음 그림과 같이 먼저 장면에서 아주 밝은 고휘도(High Brightness or

Luminescence) 부분을 추출해야 하고 이것을 이미지에 저장을 해야 합니다.

만약 6방향으로 분산하는 빛을 표현하고 싶다면 6방향으로 늘려야 합니다. 6방향에 대한 결과를 저장해야 하므로 여기서 최소한 6장의 이미지와 이 결과를 저장할 한 장의 이미지 총 7장이 필요합니다.



#### <크로스 필터 만드는 방법>

쉐이더 코드로 6방향으로 색상을 늘리게 되면 군데군데 빛이 끊어질 수 있습니다. 이것을 매우기 위해 흐림 효과를 적용합니다. 이렇게 적용하고 나서 장면과 다시 합칩니다.

전체적인 내용은 이전의 흐림 효과에 밝은 부분 추출과 6방향으로 늘려 처리하는 과정이 더 추가가 되었습니다. 이것은 그냥 하면 될 것 같기도 합니다. 그런데 여기서 하나 더 알아야 할 것이 있습니다. 그것은 밝은 부분을 더욱 강하게 저장해 놓아야 6방향으로 늘이면서 점점 빛의 세기를 줄여가며 크로스 필터 효과를 만들어야 하는데 기존의 32비트 텍스처는 각 색상의 표현이 최대 255 밖에 안됩니다. 따라서 쉐이더 코드에서 밝기를 아무리 올려 놓아도 최대 255 이상은 소용이 없게 됩니다.

이런 이유로 각 색상을 8비트 보다 더 큰 정보를 저장할 수 있는 텍스처가 필요합니다. DXSDK는 아주 높은 휘도의 색상 정보를 저장할 수 있는 각 채널당 16비트 부동 소수점 형식의 `D3DFMT_A16B16G16R16F`와 32비트 형식의 `D3DFMT_A32B32G32R32F` 포맷을 지원하고 있습니다. 이 둘 중의 하나로 텍스처를 만들어야 고휘도 처리 결과가 제대로 저장이 됩니다.

대충 어떤 형식으로 처리해야 하는지 방법을 알았으니까 본격적으로 구현해 봅시다. 먼저 각 단계별 텍스처들을 모아 보면 다음과 같이 전체 장면을 저장할 텍스처 1장, 고휘도 추출을 위한 텍스처 1 장, 6방향과 이를 저장할 텍스처 7장 이 필요한데 고휘도 추출을 먼저 Y 방향 흐림 효과에 사용할 텍스처에 저장하면 되므로 총 10장 정도의 텍스처가 필요합니다.

```
IrenderTarget* m_pTrnd;           // Rendering Target Texture for Scene
IrenderTarget* m_pTrndX;          // Rendering Target Texture for Blur X
IrenderTarget* m_pTrndY;          // Rendering Target Texture for Blur Y
IrenderTarget* m_pTrndC[6];        // Cross Texture
IrenderTarget* m_pTrndS;          // Cross Texture All
```

`IrenderTarget` 는 실시간 텍스처를 생성해 주는 클래스입니다. 이 클래스에 대한 객체를 생성할 때 다음과 같이 "HDR16" 문자열을 넣어주면 채널당 16비트 `D3DFMT_A16B16G16R16F` 포맷의 텍스처를 생성해 줍니다. 내부 코드는 [h4\\_06\\_cross1.zip](#)의 `RenderTarget.cpp`에 있으니 참고 하기 바랍니다.

```
m_fTxW = 400;
m_fTxH = 300;
LcD3D_CreateRenderTarget("HDR16", &m_pTrndX, m_pDev, m_fTxW, m_fTxH);
```

텍스처 생성은 이렇게 끝내고 다음으로 고휘도 추출하는 HLSL부분입니다.

```
float4 PxLLumi(SvsOut In) : COLORO
...
    float4 Out=0;
    float4 t0 = tex2D(smpDif, In.Tex);

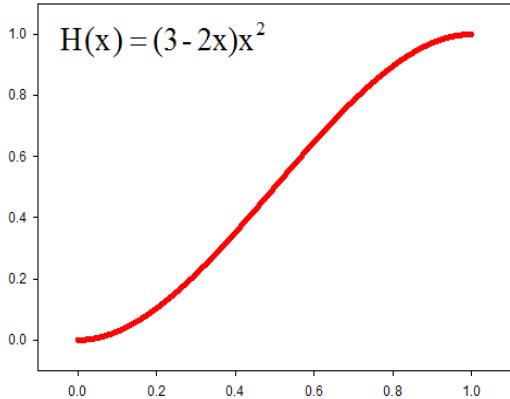
    t0.g *=1.11f;    t0.b *=0.9f;
    t0 = pow(t0,4);

    if(t0.r+t0.b+t0.b<0.95f)
        t0=0.f;

    t0 = smoothstep(0.6,0.9, t0);
    t0 = pow(t0,12)*2;
    Out = t0;
...
```

전체 장면이 파란색이 많아 녹색을 조금(1.11f) 강조했습니다. 코드 중간에 `pow()` 함수를 사용해서 색상의 밝기가 1 근처에 있는 픽셀만 남겨놓도록 하고 있습니다.

`smoothstep()`함수는 세 인수, `min`, `max`, 색상을 받아서 색상이 `min` 보다 작거나 같으면 0을, `max` 보다 크거나 같으면 1을 반환 합니다. 그 사이에 있는 함수는 그림처럼 에르미트 보간(Hermite Interpolation)을 하는 함수입니다.



```
float smoothstep(float min, float max, float x)
{
    x = saturate((x - min)/(max - min));
    return x*x*(3-2*x);
}
```

saturate:  $x < 0 \rightarrow x = 0$ ,  $x > 1 \rightarrow x = 1$

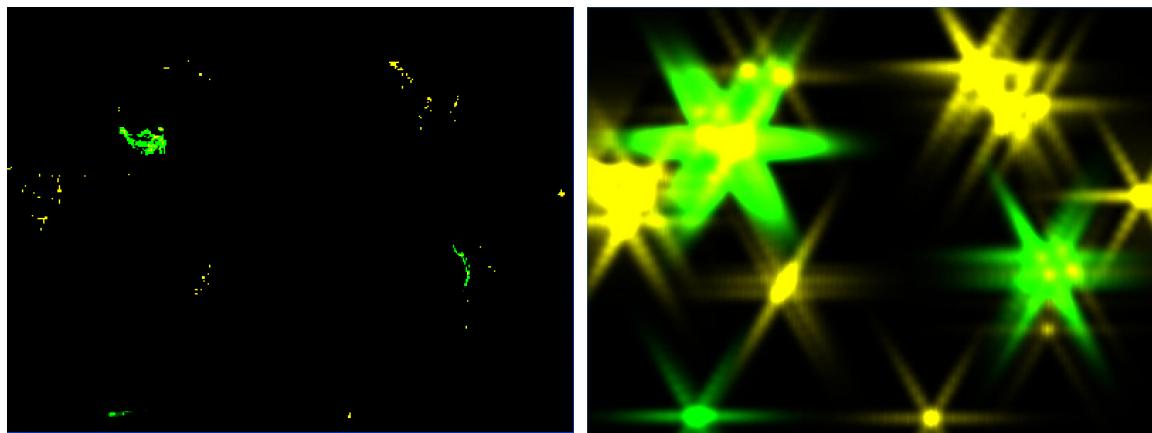
< Hermite Interpolation >

o 쉐이더 코드는 CShaderEx::FrameMove()에서 2번째 고크도 추출에서 호출됩니다.

```
INT CShaderEx::FrameMove()
...
// 2. 축소된 텍스처에 Luminescence가 강한 부분을 그린다.
m_pTrndY->BeginScene();
...
hr = m_pDev->SetVertexDeclaration(m_pFVF);
...
hr = m_pEft->Begin(NULL, 0);
hr = m_pEft->Pass(0);
hr = m_pDev->DrawPrimitiveUP();
...
...
```

6방향으로 늘리는 작업은 쉐이더코드의 Px1Star() 함수에서 합니다. for문 안에서 각 방향에 따라 샘플링을 하고 이 값을 계속 누적해 나갑니다.

```
float4 Px1Star(SvsOut In) : COLORO
...
for(int i=0; i<MAX_SAMP; ++i)
{
    uv = In.Tex + float2(m_StarVal[i].x, m_StarVal[i].y);
    Out += tex2D(smpDif, uv) * m_StarVal[i].z;
}
...
```



<고희도 검출, 6방향 늘리기. 하드웨어가 팬찮다면 방향 늘리기 전 흐림 효과를 1~2회 적용한다>

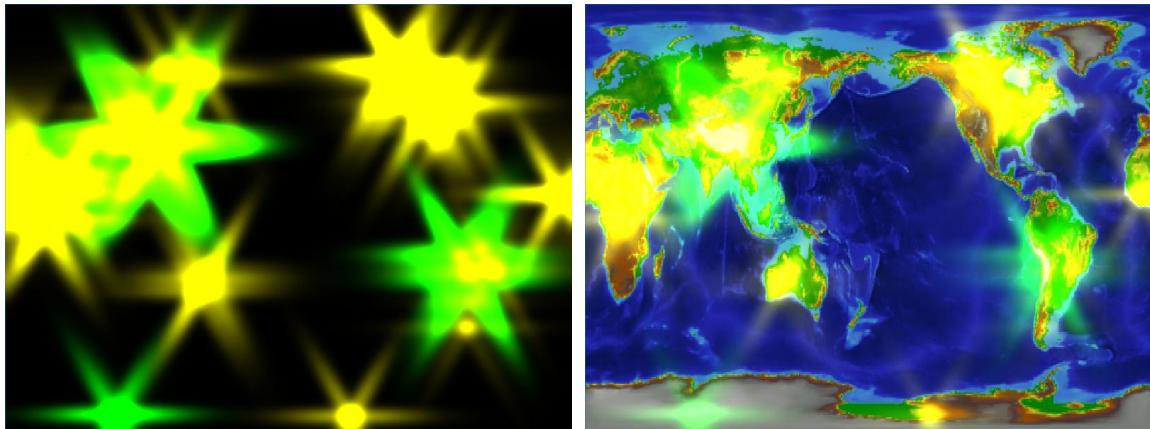
6방향으로 픽셀을 늘리고 다음으로 흐림 효과를 적용해서 중간에 끊길지도 모르는 부분을 채워 줍니다.

마지막으로 6장의 빛을 모아 하나의 텍스처에 저장합니다.

```
float4 Px1StarAll(SvsOut In) : COLOR0
...
    float4 Out=0;
    Out += tex2D(s0, In.Tex);
...
    Out += tex2D(s5, In.Tex);
```

이 저장된 텍스처와 마지막에 전체 장면과 합칩니다.

```
float4 Px1All(SvsOut In) : COLOR0
{
    float4 Out=0;
    float4 t0 = tex2D(smpDif, In.Tex);
    float4 t1 = tex2D(smpDif2, In.Tex);
    Out = t0*.8f + t1*.3f;
    Out.w = 1;
    return Out;
}
```

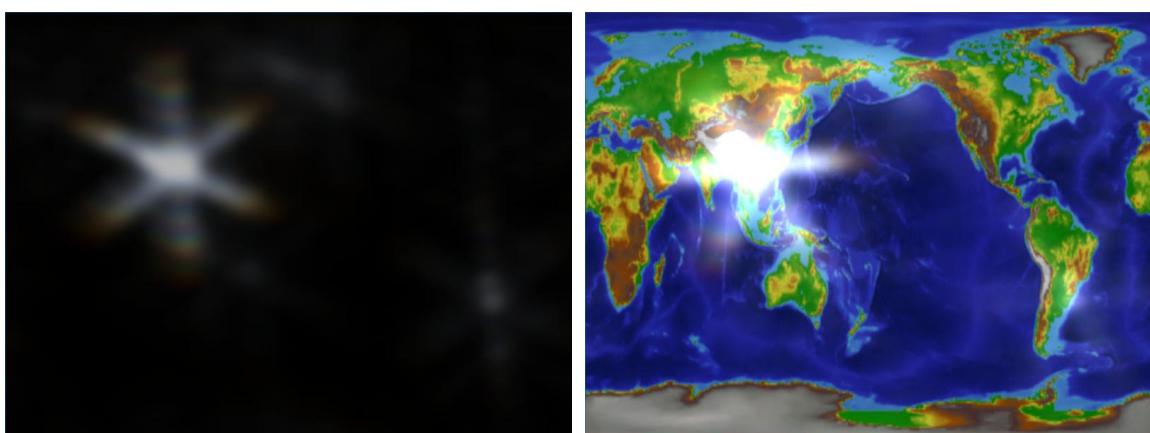
<크로스 필터 효과: [h4\\_06\\_cross1.zip](#)>

빛은 파장에 따라 굴절 되는 각도가 다릅니다. 크로스 필터에도 이것을 간단하게 적용하는 방법은 R, G, B에 따라 샘플링 되는 지점의 U, V에 조금씩 차이를 주면 색수차(chromatic aberration)를 만들어 낼 수 있습니다.

다음은 붉은 색에 대한 색수차 적용입니다.

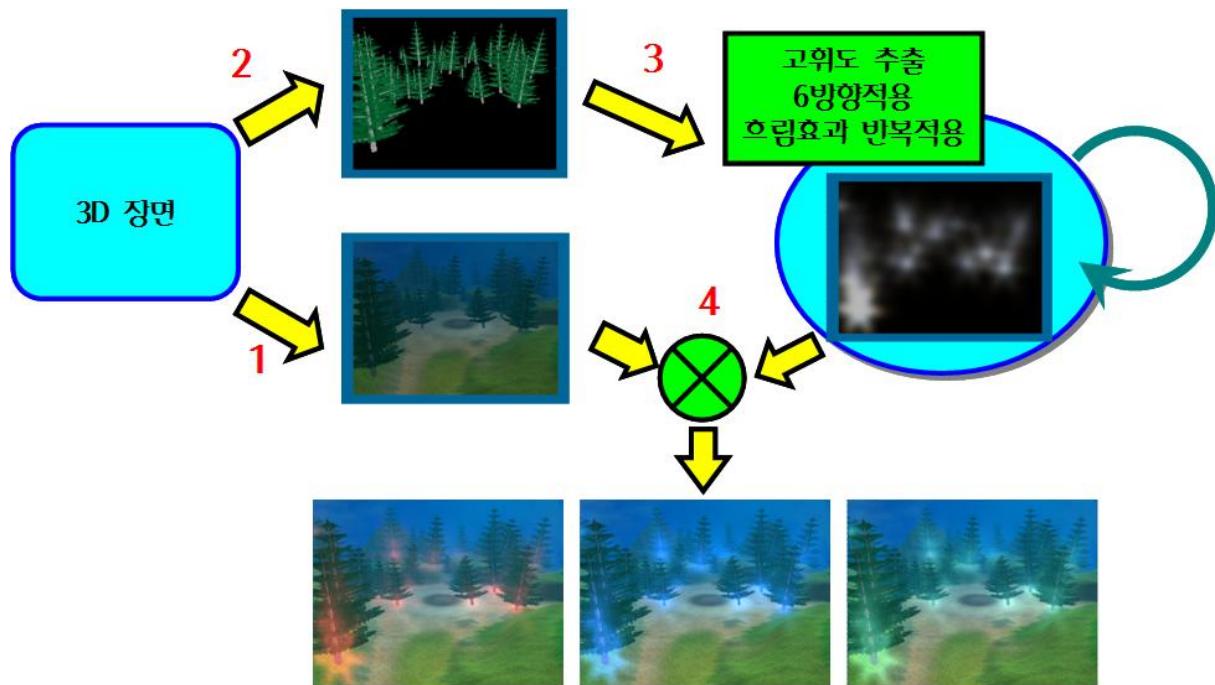
```
float4 Px1StarR(SvsOut In) : COLORO
...
for(int i=0; i<MAX_SAMP; ++i)
{
    u = In.Tex.x + m_StarVal[i].x*1.25f;
    v = In.Tex.y + m_StarVal[i].y*1.25f;
    Out.r += tex2D(smpDif, float2(u,v)) * m_StarVal[i].z;
}
...
```

이것을 녹색, 파란색에도 적용하면 다음과 같은 화면을 얻을 수 있습니다.



<색수차가 적용된 크로스필터: [h4\\_06\\_cross2.zip](#)>

이 정도 연습했으면 3D 장면에 적용해볼 차례입니다. 작업은 다음 그림처럼 진행이 될 것입니다.



<크로스 필터: 3D 장면 적용>

지난 시간에 파티클을 이용해서 화면의 왜곡을 만들어 본 적이 있습니다. 그 때에 왜곡시키는 요소를 따로 텍스처에 저장을 하고 해당 효과를 적용했습니다. 지금도 마찬가지로 3D 장면과 휘도를 적용 대상을 따로 렌더링 해야 합니다.

여기서는 특별한 오브젝트 추가 없이 나무가 고휘도를 만드는 오브젝트라 가정하고 이 들을 가지고 크로스 필터 효과를 만들어 보겠습니다. 먼저 1번 단계의 전체 장면과 고휘도 대상인 2번의 나무만 렌더링을 각각 진행 합니다. 2번 나무만 렌더링 한 텍스처를 가지고 크로스 필터를 만듭니다. 4번 단계에서 이 둘을 혼합 합니다.

CMain 클래스에서 전체 장면과 휘도 장면을 저장하기 위해 다음 2개의 멤버를 선언합니다.

```
IrenderTarget* m_pTrnd0;      // 전체 장면
IrenderTarget* m_pTrnd1;      // 고휘도 적용 장면
```

전체 장면은 화면 크기대로 만들고 휘도로 사용할 텍스처는 렌더링 속도 때문에 축소해서 만듭니다.

```

HRESULT CMain::Init()
...
// 3D 장면 저장 텍스처 생성
LcD3D_CreateRenderTarget(NULL, &m_pTrnd0, m_pd3dDevice);
...
// 휘도 저장 텍스처 생성
FLOAT fTexW = 256;
FLOAT fTexH = 256;
LcD3D_CreateRenderTarget(NULL, &m_pTrnd1, m_pd3dDevice, fTexW, fTexH);

```

이들을 매 프레임마다 렌더링을 합니다. 그리고 CShaderEx 클래스 객체에 이 두 텍스처를 전달합니다.

```

HRESULT CMain::FrameMove()
...
// Rendering Target에 장면을 그린다.
m_pTrnd0->BeginScene();
    m_pd3dDevice->Clear(...);
    RenderScene(0);
m_pTrnd0->EndScene();

// 휘도 부분에 대한 장면을 그린다.
m_pTrnd1->BeginScene();
    m_pd3dDevice->Clear(...);
    RenderScene(1);
m_pTrnd1->EndScene();
// CShaderEx 클래스에 장면, 휘도 텍스처 연결
LPDIRECT3DTEXTURE9 pTx0 = (LPDIRECT3DTEXTURE9)m_pTrnd0->GetTexture();
LPDIRECT3DTEXTURE9 pTx1 = (LPDIRECT3DTEXTURE9)m_pTrnd1->GetTexture();
m_pShader->SetSceneTexture(pTx0);
m_pShader->SetLuminescenceTexture(pTx1);
SAFE_FRMOV(m_pShader);
...

```

CMain에서 할 일은 끝났습니다. 다음으로 CShaderEx 클래스의 과정입니다. CShaderEx 클래스에서는 CMain에서 전달한 3D 장면과 휘도에 대한 텍스처 포인터 2개를 가지고 있어야 합니다. 다음으로 크로스 필터에 대한 텍스처 9장을 준비합니다. 이것은 순서를 잘 맞추면 더 줄일 수 있습니다.

일단 여기서는 각 장면이 제대로 처리되는지 확인을 위해 넉넉하게 사용하겠습니다.

```
PDTX      m_pTexS;           // Scene Texture
PDTX      m_pTexL;           // Luminescence Texture
IrenderTarget* m_pTrndX;     // Rendering Target Texture for Blur X
IrenderTarget* m_pTrndY;     // Rendering Target Texture for Blur Y
IrenderTarget* m_pTrndC[6];   // Cross Texture
IrenderTarget* m_pTrndS;     // Cross Texture All
```

적당한 크기를 할당해 텍스처를 만들고 매 프레임마다 크로스 필터 효과를 만들어 냅니다. 이전과 달라진 부분은 1번에서 회도용 텍스처를 사용해야 하는 것과 크로스 모양을 만들기 전에 동적인 움직임을 주기 위해 약하게 시간에 따라 별 모양을 회전 시켰습니다.

```
INT CShaderEx::FrameMove()
...
// 1. Luminescence Texture를 사용한다.
pTx = m_pTexL;
...
// 3. 축소된 텍스처를 뭉갠다.
...
D3DXMATRIX mtViw;
m_pDev->GetTransform(D3DTS_VIEW, &mtViw);
mtViw._41 = 0;    mtViw._42 = 0;    mtViw._43 = 0;

D3DXQUATERNION q;
FLOAT fTime=GetTickCount()*0.001f;
D3DXQuaternionRotationMatrix(&q, &mtViw);
float fc= acosf(q.w*0.99f) * 24.f + fTime*0.1f;

//4. Cross를 만든다.
for(j=0; j<6; ++j)
...
    float fTheta = (fc+2.f * j * D3DX_PI)/6;
...
```

이전 코드를 거의 그대로 사용하지만 각 단계별로 원하는 장면이 나오는지 중간중간 점검합니다. CShaderEx::Render() 함수의 #if 1 을 #if 0 으로 하고 주석 처리된 부분을 다시 활성화 시키면

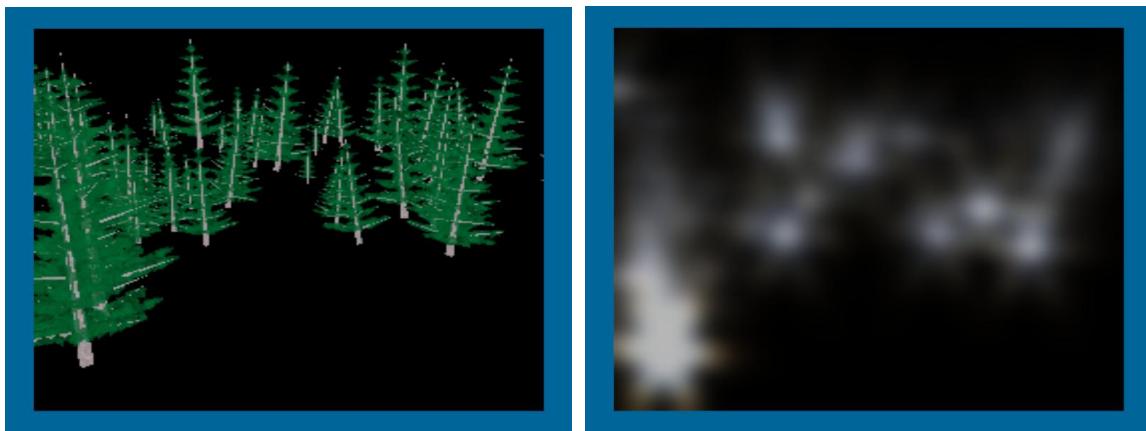
각 단계별 장면을 화면에 출력합니다.

```
// 전체 장면
//m_pDev->SetTexture(0, m_pTexS);
//m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(VtxDUV1));

// 고휘도 장면
//m_pDev->SetTexture(0, m_pTexL);
//m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(VtxDUV1));

// 크로스 필터 장면
pTex = (PDTX)m_pTrndY->GetTexture();
m_pDev->SetTexture(0, pTex);
m_pDev->DrawPrimitiveUP( D3DPT_TRIANGLEFAN, 2, pVtx, sizeof(VtxDUV1));
```

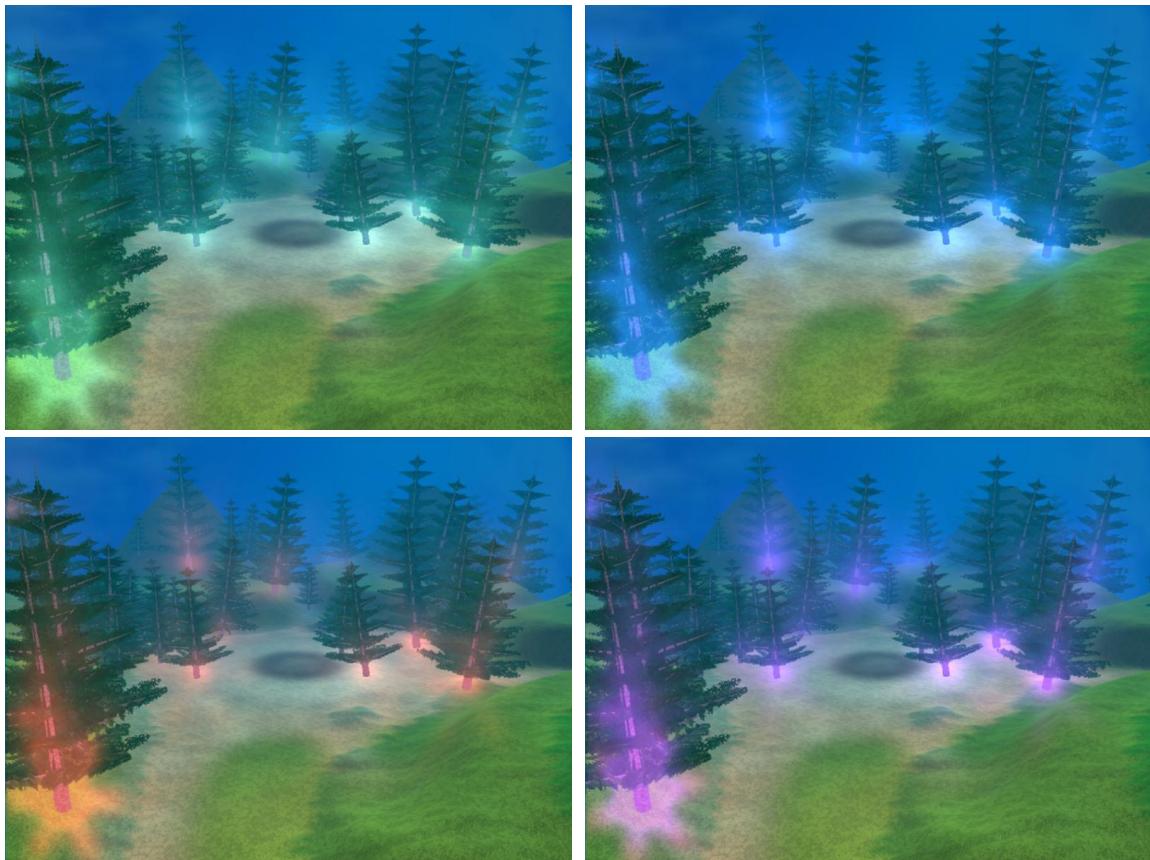
다음 그림은 각 단계별 장면입니다.



<크로스 필터 휘도 추출: [h4\\_06\\_cross2\\_a.zip](#)>

쉐이더 파일 "hlsl.fx" 파일에서 장면과 크로스 필터를 섞는 Px1All() 함수의 마지막 단계에서 불은색, 초록색, 파란색의 비율을 다르게 적용하면 다음 그림과 같은 장면을 만들어냅니다.

```
float4 Px1All(SvsOut In) : COLOR0
...
//      Out = t0*1.5f + t1*.4f*float4(1.5, 0.3, .2, 1);
//      Out = t0*1.5f + t1*.4f*float4(.6, 1., .8, 1);
//      Out = t0*1.5f + t1*.4f*float4(.3, 0.7, 1.6, 1);
      Out = t0*1.5f + t1*.4f*float4(1.3, .3, 1.4, 1);
...
```



<3D 장면에서의 크로스 필터: [h4\\_06\\_cross2\\_a.zip](#)>

## 6.6 수목화 효과

수목화 효과는 간단하게 설명하면 흐림 효과와 외곽선을 혼합해서 표현합니다. 순서대로 외곽선 추출 방법을 알아보고 다음으로 이전에 사용했던 흐림 효과를 적용해서 수목화 효과를 만들어 보겠습니다.

### 6.6.1 외곽선 추출

앞의 흐림 효과(Blur Effect)가 수학으로 말하면 색상에 대한 적분이라 할 수 있습니다. 즉, 주변의 색상을 해서 평균을 내는 것입니다. 외곽선 추출을 반대로 미분에 해당합니다. 미분은 간단히 정의하면 변화 량( $\Delta$ )이라 할 수 있습니다. 예를 들어 색상을 [0, 255]로 표현할 때 회색 10과 회색 50은 변화 량이 40이지만 회색 250과 회색 248은 변화 량이 2입니다.

단순히 색상만 가지고 화면에 출력하면 회색 250과 248은 거의 하얀색으로 나타나고 회색 10과 20은 거의 검정색으로 나오겠지만 변화 량을 화면에 표현하면 10과 20이 더 밝게 나올 것이라는 것

은 당연합니다.

그럼 이 변화 량은 어떻게 만들어야 할까요?

가장 빠른 방법은 마스크를 이용하는 것입니다. 마스크는 하나의 픽셀에 대하여 이웃한 픽셀 들에 대해서 정방 행렬 형태로 가중치를 지정한 값으로 외곽선 추출에서는 소벨(Sobel), 라플라시안(Laplacian) 마스크를 주로 사용합니다.

다음 그림은 소벨 마스크의 X, Y 방향 마스크와 라플라시안 마스크입니다.

-1	-2	-1	-1	0	1	-1	-1	-1
0	중심 픽셀(0)	0	-2	중심 픽셀(0)	2	-1	중심 픽셀(8)	-1
1	2	1	-1	0	1	-1	-1	1

<소벨(Sobel), 라플라시안(Laplacian)>

소벨 마스크는 윤곽선 검출의 가장 대표적인 마스크로 X, Y 두 방향에 대한 필터가 있어 이들을 각각 적용한 후에 최종 기울기(Gradient)를 결정합니다. 이 기울기를 수식으로 표현하면 다음과 같습니다.

$$G = \left( (\sum \Delta x_i)^2 + (\sum \Delta y_i)^2 \right)^{1/2}$$

앞의 그림의 마스크 값을 이용해서 쇼이더 코드로 작성한다면 다음과 같습니다.

```
float4 Px1Sobel(SvsOut In) : COLOR0
{
    float2 Offset[8] =
    {
        {-1,-1}, { 0,-1}, { 1,-1},
        {-1, 0}, { 1, 0},
        {-1, 1}, { 0, 1}, { 1, 1},
    };

    float4 Out=float4(0,0,0,1);
    float4 Mon=float4(0.299, 0.587, 0.114, 0);
    float2 uv;
```

```

float4 Tex[8];
float4 TexVert;
float4 TexHorz;

for(int i =0; i < 8; i++)
{
    Offset[i].x /= m_TexW;
    Offset[i].y /= m_TexH;

    uv = In.Tex + Offset[i];
    Tex[i] = tex2D( smpDif, uv);
    // convert to Mono
    Tex[i] = dot(Tex[i], Mon);
}

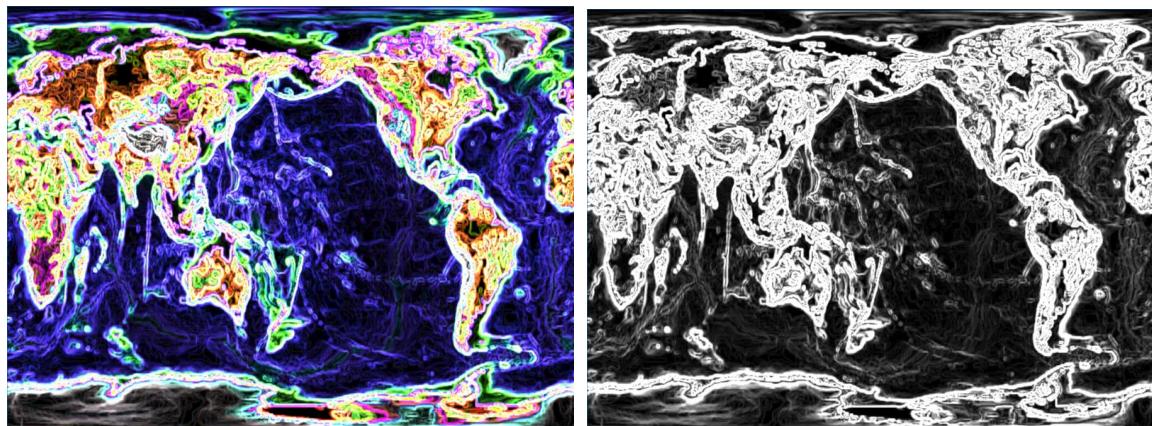
// Vertical
TexVert = -(Tex[0] + Tex[5] + 2*Tex[3]);
TexVert += (Tex[2] + Tex[7] + 2*Tex[4]);

// Horizontal
TexHorz = -(Tex[0] + Tex[2] + 2*Tex[1]);
TexHorz += (Tex[5] + Tex[7] + 2*Tex[6]);

Out = sqrt( TexHorz*TexHorz + TexVert*TexVert );
Out = saturate(Out);
Out *= 4;
Out.w = 1.f;
return Out;
}

```

o) 쉐이더 코드를 적용하면 다음과 같은 장면을 얻을 수 있습니다.



&lt;소벨(Sobel) 마스크에 의한 윤곽선&gt;

소벨 마스크와 함께 또한 가장 많이 사용되는 마스크는 라플라시안(Laplacian) 입니다. 이 마스크는 이론적으로 2차 미분 연산자 사용하며 한 번에 하나의 마스크로 윤곽선 검출 수행 연산 속도가 매우 빠르다는 것이 가장 큰 장점입니다.

결과는 다른 연산자와 비교하여 날카로운 윤곽선을 만들어 내는 것이 특징입니다. 변화에 대한 (G: Gradient)은 각 변화에 대한 합으로 다음과 같이 간단하게 계산합니다.

$$G = \sum \Delta_i$$

쉐이더 코드는 다음과 같이 작성합니다.

```
float4 PxLaplacian(SvsOut In) : COLOR0
{
    float4 Mono={0.299, 0.587, 0.114, 0};
    float3 Laplacian[9] =
    {
        {-1,-1, -1}, { 0,-1, -1}, { 1,-1, -1}, // offset X, offset y, Weight
        {-1, 0, -1}, { 0, 0,  8}, { 1, 0, -1},
        {-1, 1, -1}, { 0, 1, -1}, { 1, 1, -1},
    };
    float4 Out=float4(0,0,0,1);
    float4 txC=0;
    float2 uv;
    float x, y, w;

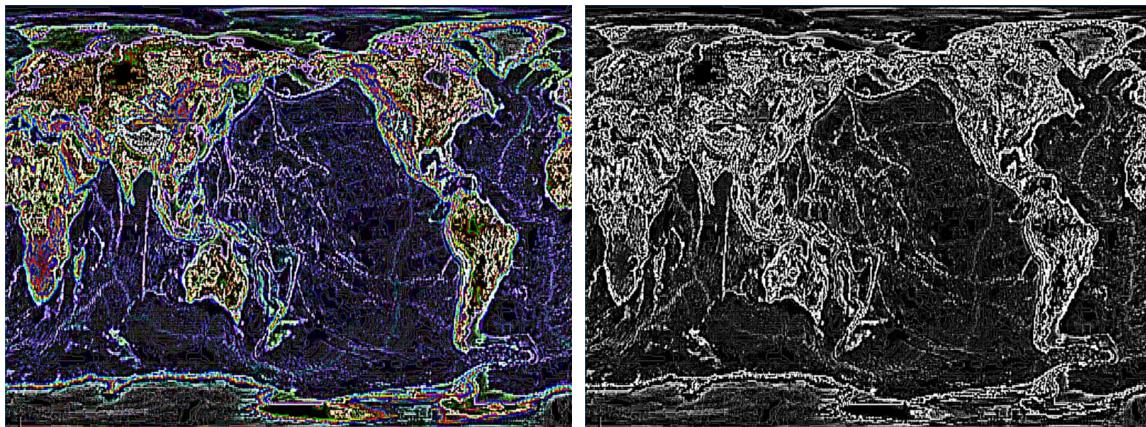
    for(int i=0; i<9; ++i)
```

```

{
    x = Laplacian[i].x/m_TexW;
    y = Laplacian[i].y/m_TexH;
    w = Laplacian[i].z;
    uv = In.Tex + float2(x, y);
    txC+= tex2D(smpDif, uv)*w;
}

float d=dot(txC, Mono);
d *=30;
Out.xyz = d;
return Out;
}

```



<라플라시안(Laplacian) 마스크에 의한 윤곽선>

이렇게 외곽선 추출은 이웃한 색상의 변화 량(Gradient)를 조사해서 이 값의 크기를 색상으로 결정하는 것입니다. 외곽선 추출이 이런 내용이라면 마스킹(Masking) 값을 이용하지 않고 직접 변화 량을 계산해서 적용해 볼 수 있습니다. 지금 소개하려는 방법은 색상 r, g, b를 3차원 좌표  $x, y, z$ 처럼 생각해서 두 색상의 변화 량을  $\Delta_i = \sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2 + (z_0 - z_i)^2}$ 으로 계산해 보자는 것입니다. 이 방법은 앞의 방법보다 제곱 근 계산이 들어가서 무겁지만 본래의 의도에 가장 충실한 방법이 될 것입니다.

$$G = \sum \sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2 + (z_0 - z_i)^2}$$

이것은 다음과 같이 쉐이더 코드로 쉽게 표현할 수 있습니다.

```

float4 Px1Dist(SvsOut In) : COLOR0
{
    float4 Out=float4(0,0,0,1);
    float2 uv;
    float3 txI;
    float3 txC;
    float D=0;

    txI = tex2D(smpDiff, In.Tex);

    for(int j=-1; j<=1; ++j)
    {
        for(int i=-1; i<=1; ++i)
        {
            if(!(0==i && 0==j))
            {
                uv = In.Tex + float2(i/m_TexW, j/m_TexH);
                txC      = tex2D(smpDiff, uv);
                txC      -= txI;
                D        += length(txC);
            }
        }
    }

    D *=3;
    if(D>1.)
        Out.rgb = 1;

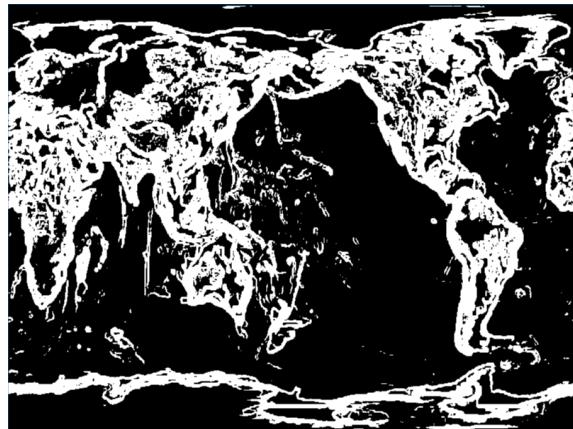
    return Out;
}

```

코드에서 먼저 중심 픽셀의 색상을 `txI = tex2D(smpDiff, In.Tex)` 으로 구한 후에 `for` 문에서 `length()`함수로 색상 차이에 대한 길이를 구하고 있습니다. `length()`함수는 내부에서 `sqrt()`함수를 호출해서 길이를 계산하는 함수입니다. 코드의 내용은 두 픽셀의 색상차이를 길이로 저장하고 있습니다.

이 쇄이더 코드를 가지고 실행하면 다음과 같이 색상의 변화에서 이전의 방식 보다 좀 더

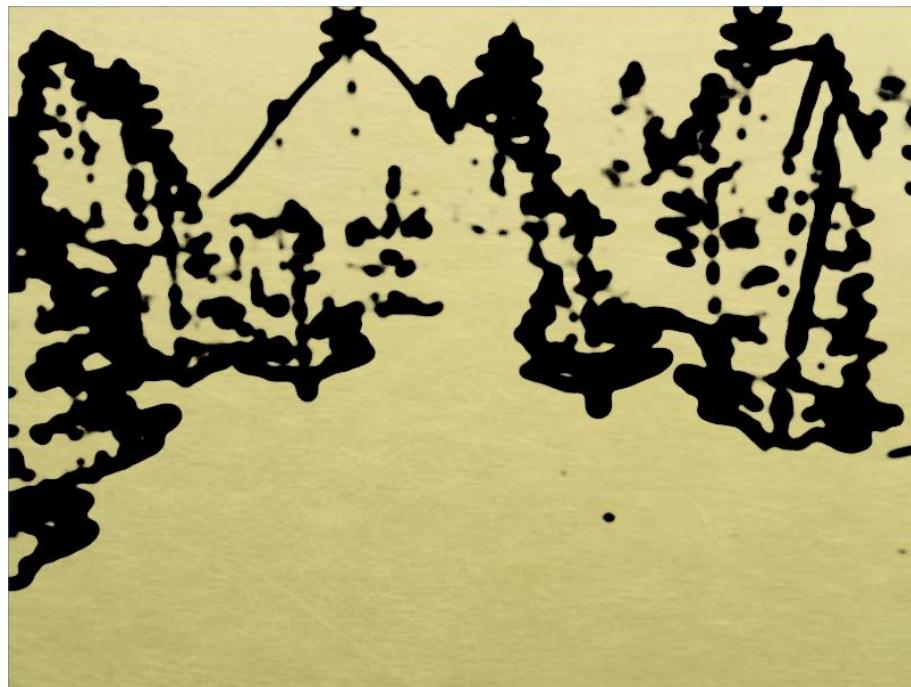
뚜렷하게 만들어 갈 수 있습니다.



<색상의 거리를 이용한 외곽선 추출>

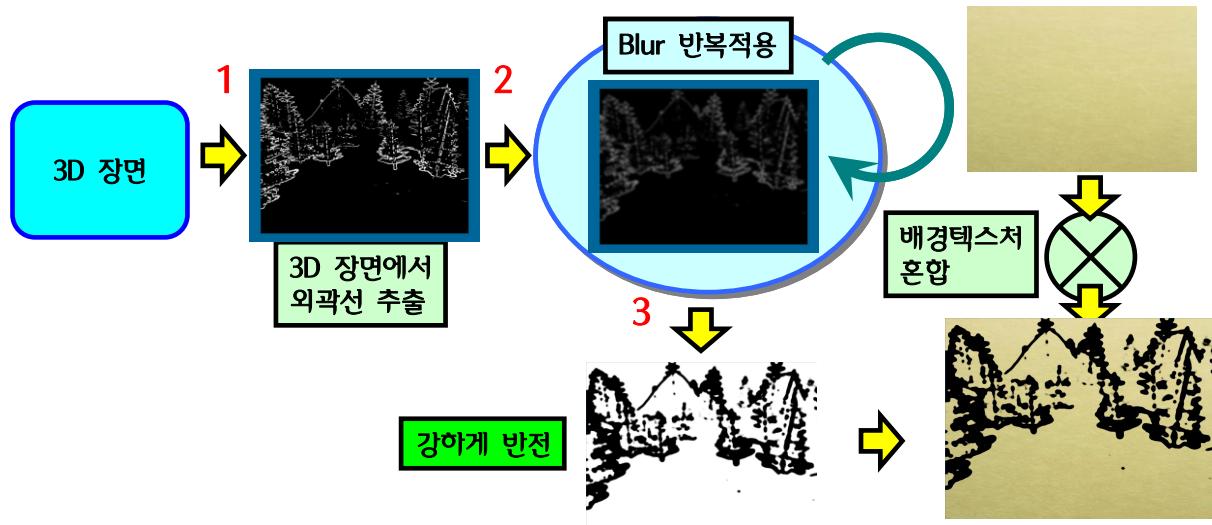
예제 [h4\\_07\\_out\\_line1.zip](http://3dapicom.com/h4_07_out_line1.zip) 를 실행하면 처음에는 거리를 이용한 외곽선 추출을 볼 수 있습니다. 다음으로 'S' 또는 'L' 키를 누르면 각각 소벨 마스크, 라플라시안 마스크를 적용한 외곽선 추출을 볼 수 있습니다.

이러한 외곽선 추출은 카툰 쇼이딩, 수목화, 색상의 영역 설정 등에서 두루 사용이 됩니다. 다음의 [h4\\_07\\_out\\_line2.zip](http://3dapicom.com/h4_07_out_line2.zip) 는 완벽한 수목화 기법은 아니지만 외곽선만 조정해서 수목화 느낌을 살린 예제입니다.



<외곽선을 이용한 수목화 [h4\\_07\\_out\\_line2.zip](http://3dapicom.com/h4_07_out_line2.zip)>

이 장면은 다음 그림과 같은 과정을 통해서 만든 것입니다.



<[h4\\_07\\_out\\_line2.zip](#) 예제 제작 과정>

3D 장면을 저장한 텍스처에서 외곽선을 추출합니다. 여기서 사용한 외곽선은 거리를 이용한 방법입니다. 외곽선을 그대로 사용하면 좁고, 날카롭습니다. 봇 터치 느낌을 만들기 위해 흐림 효과를 반복 적용합니다. 흐림 효과를 반복 적용하면 밝기가 많이 낮아집니다. 따라서 강하게 색상을 반전 시키면 먹물 느낌을 만들어 낼 수 있습니다. 준비한 배경 텍스처와 혼합하면 앞의 장면을 만들어 낼 수 있습니다.

[h4\\_07\\_out\\_line2.zip](#)의 CShaderEx::FrameMove() 함수에서 원 장면에서 외곽선을 추출하고 흐림 효과를 적용하고 있습니다. CShaderEx::Render() 함수에서는 강하게 반전(Inversion)한 후에 배경 텍스처와 혼합하는 작업을 하고 있습니다.

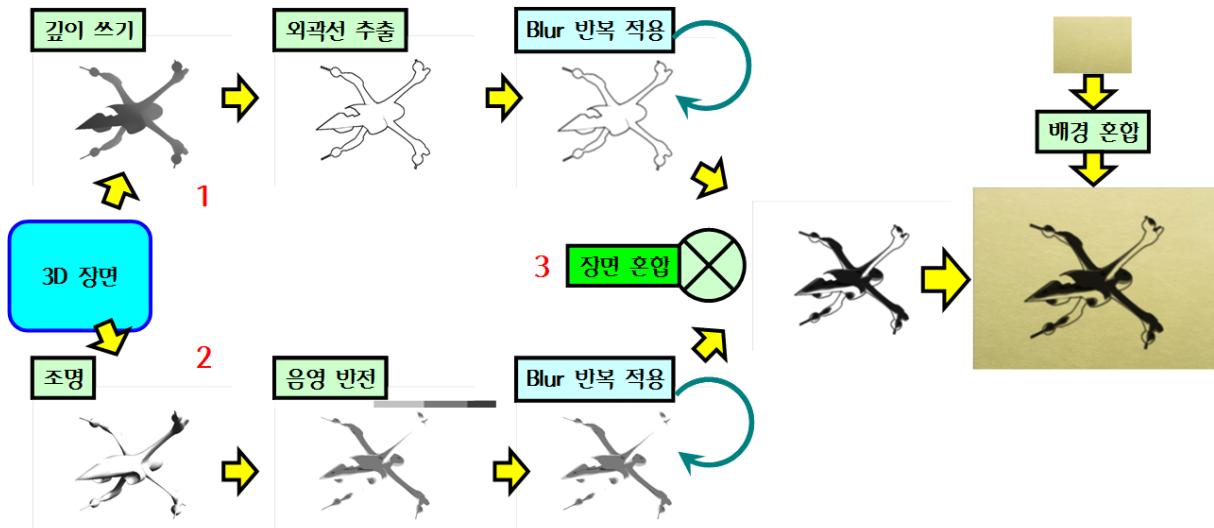
## 6.6.2 수목화 렌더링

수목화 렌더링 기법은 조명과 외곽선 추출을 이용한 방법입니다. 보통 조명에서 밝은 부분은 밝게 어두운 부분은 어둡게 처리하지만 수목화에서는 이와 반대로 밝은 부분은 어둡게 어두운 부분은 밝게 처리합니다. 이렇게 해야만 먹물로 그림을 그린 느낌을 만듭니다. 봇에 의한 번짐을 표현하기 위해서 밝고 어둠의 반전을 만든 후에 흐림 효과를 적용합니다. 이렇게 만든 것을 텍스처에 저장을 합니다.

다음으로 외곽선을 렌더링 오브젝트에서 추출합니다. 이 때 깔끔한 외곽선을 만들기 위해서 색상

의 변화를 가장 잘 만들 수 있는 오브젝트의 법선 값, 또는 깊이 값을 색상으로 저장해서 이 저장된 텍스처에서 외곽선을 추출합니다. 추출한 텍스처를 역시 흐림 효과를 적용 합니다. 마지막으로 앞서 반전을 이용해 만든 구한 텍스처와 외곽선을 만든 텍스처와 혼합합니다.

이 과정을 그림으로 표현하면 다음과 같습니다.



<수목화 렌더링 과정>

앞에서 인접한 색상의 차이가 클 수록 외곽선이 잘 표현된다는 것을 알았습니다. 주변 색상과 의도적으로 차이를 만들기 위해 오브젝트의 깊이 값 또는 법선 벡터를 텍스처에 저장해서 사용하는데 여기서는 깊이 값을 사용하도록 하겠습니다.

깊이 값을 저장하는 방법은 간단합니다. 먼저 정점 처리과정에서 위치에 대한 변환 후의 값을 텍스처 좌표 값으로 저장합니다. 다음으로 픽셀 처리과정에서 쉐이더로 이 값을 실시간으로 만든 텍스처에 저장합니다.

깊이 값을 저장하기 위해서 정점 쉐이더는 다음과 같이 작성합니다.

```
float4x4 m_mtWVP; // World * View * Projection
...
// 베텖스 쉐이더 출력
struct SvsOut
{
    float4 Pos : POSITION;
...
}
```

```

float4 Nrp : TEXCOORD7;           // Normal vector+ 깊이 값
};

SvsOut InkVtx(float3 Pos : POSITION)
{
    SvsOut Out = (SvsOut)0;
    ...
    float4 P = mul(float4(Pos, 1), m_mtWVP);           // 정점 위치 변환
    ...
    Out.Nrp.w      = P.z;
    return Out;
}

```

보통 정점 처리 과정에서 만든 데이터를 픽셀 쉐이더로 보낼 때 참조가 안 되는 값들은 텍스처 좌표 데이터(TEXCOORD7~0)를 선택해서 사용합니다. 예를 들어 정점의 위치를 픽셀 쉐이더로 전달할 수 있지만 값은 읽지 못합니다. 이런 이유로 앞의 쉐이더 코드는 변환 후의 정점의 깊이 값을 7번 인덱스 텍스처 좌표의 w 값에 저장한 것입니다. 또한 깊이 값만 사용하고 있어서 새로운 텍스처 인덱스를 부여하면 다른 곳에서 사용할 수 있는 폭이 줄어 들기 때문입니다.

픽셀 쉐이더는 정점의 깊이 값을 텍스처에 쓰기만 하면 될 것 같지만 변환 후의 깊이 값은 거의 1근처에 몰려 있습니다. 따라서 이 값을 적당한 값으로 나누어야 하는데 대략 뷰 체적(View Volume) 때 Far 평면까지의 거리 값을 사용해도 되고, 색상의 범위가 [0, 255] 이므로 이것의 2 배 정도 되는 500 정도 되는 값을 사용해도 됩니다.

```

float4 InkPx1Depth(SvsOut In) : COLOR0
{
    float4 Out = 1;
    float4 Nrp = In.Nrp;
    float d;

    d = Nrp.w;
    d /= 500.f;
    Out.xyz = d;
    return Out;
}

```

깊이 값을 텍스처에 저장할 수 있으니 이제 외곽선 만드는 것이 남았습니다. 이 과정은 예제

[h4\\_07\\_out\\_line2\\_ink1\\_edge.zip](#)에 순서대로 구현되어 있습니다.

먼저 깊이 값을 저장하고 이것을 외곽선으로 사용하기 위해서 다음과 같이 깊이 값을 텍스처와 임시 버퍼를 준비합니다.

```
IrenderTarget* m_pTrndEdg;           // Silhouette Edge Texture for Scene
IrenderTarget* m_pTrndTmp;           // Blurring for Temp
```

이들은 화면 크기와 동일하게 만듭니다. 다음으로 깊이 값을 저장해야 합니다. CShaderEx::FrameMove() 함수의 중간에 보면 다음과 같은 코드로 임시 버퍼에 깊이 값을 저장하고 있는 것을 볼 수 있습니다.

...

// 텍스처에 깊이 값을 임시 버퍼에 쓴다.

```
m_pTrndTmp->BeginScene((0x1L|0x2L), 0xFFFFFFFF);
    hr = m_pDev->SetVertexDeclaration( m_pFVFN );      // 정점 선언
    hr = m_pEft->SetMatrix( "m_mtWVP", &m_WVP);       // 월드*뷰*프로젝션 변환 행렬
```

...

```
    hr = m_pEft->SetTechnique( "TechInk");
```

...

```
    hr = m_pMesh->DrawSubset( 0 );
```

...

```
m_pTrndTmp->EndScene();
```

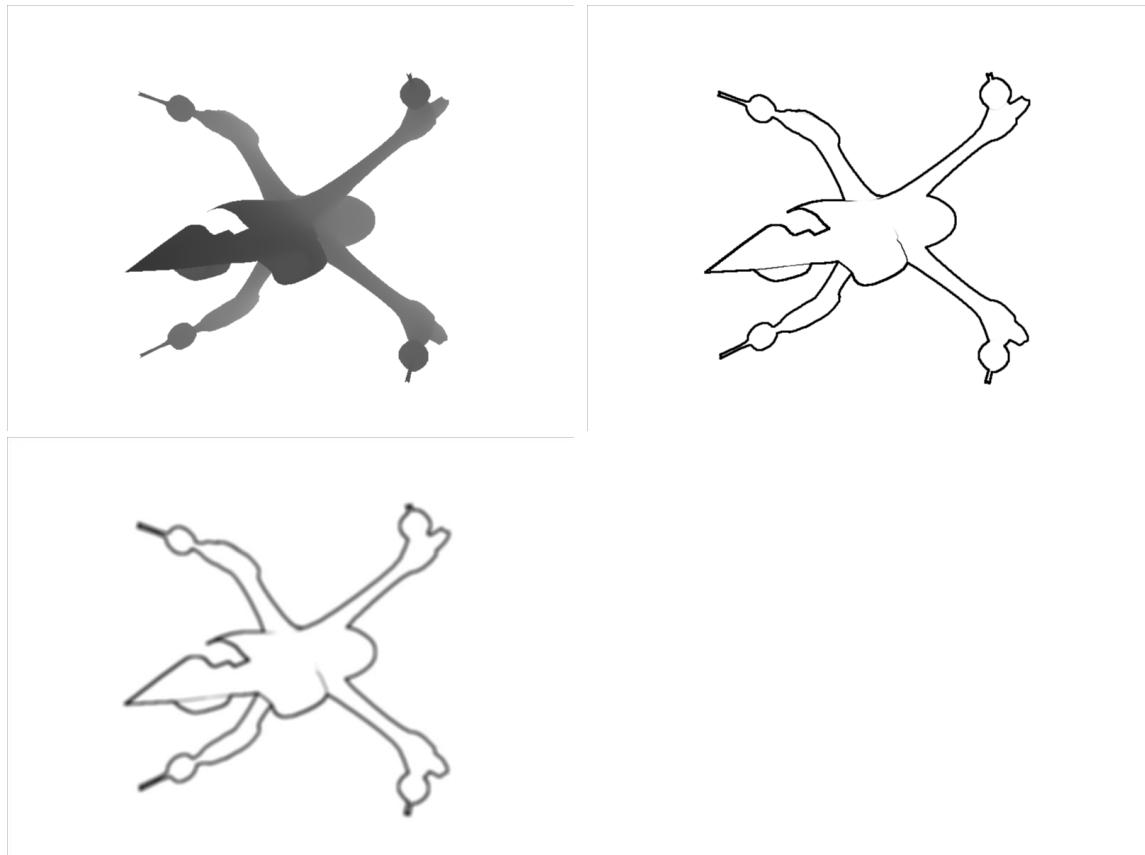
// Silhouette , 외곽선을 추출한다.

```
pTx = (PDTX)m_pTrndTmp->GetTexture();
m_pTrndEdg->BeginScene();
    hr = m_pDev->SetVertexDeclaration( m_pFVFD );
    hr = m_pEft->SetTechnique( "TechInk");
    hr = m_pDev->SetTexture( 0, pTx);
...
    hr = m_pDev->DrawPrimitiveUP(...);
...
m_pTrndEdg->EndScene();
```

코드의 내용은 먼저 임시 버퍼에 오브젝트의 깊이 값을 기록한 후에 외곽선을 추출하고 있습니다. 외곽선 추출은 "hlsl.fx"의 InkPixelEdge() 함수로 구현되어 있는데 소벨(Sobel) 마스크를 이용했는데 이 부분의 설명은 앞의 외곽선 추출을 참고 하기 바랍니다. 이렇게 외곽선 추출이 끝나면 이를

그대로 사용해도 되나 먹물의 번짐 효과를 고려해서 흐림 효과를 적용합니다.

임시 베피에 기록된 깊이 값과 외곽선 추출 후, 흐림 효과 적용에 대한 화면 출력은 다음과 같습니다.



<[h4\\_07\\_out\\_line2\\_ink1\\_edge.zip](#): 깊, 외곽선 추출, 흐림 효과 적용>

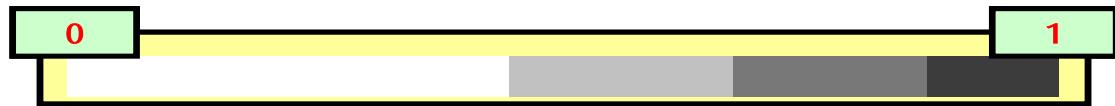
2단계의 몸체에 대한 처리는 조명의 반사에 대한 밝기를 이용합니다. 일반적으로 3D에서 밝은 부분은 밝게 처리하지만 수목화에서는 이를 반대로 처리해서 밝은 부분에는 먹물을 진하게 적용하고 어두운 부분은 얇게 처리합니다.

반사의 밝기는 Lambert 분산 값과 Phong 반사 값을 더해서 사용하지만 수목화에서는 이 둘을 곱해 버립니다. 이렇게 되면 밝은 부분과 어두운 부분의 차이가 극대화되어 부수적으로 여백의 미(美)도 어느 정도 표현이 됩니다.

이 정도의 지식만으로 몸체에 대한 색 처리를 할 수 있는데 여기에 하나 더 카툰(Cartoon) 쇼이딩에서 사용한 기법을 적용하면 농담의 변화가 연속이 아닌 이산(Discrete) 값으로 나타나 봇 느낌을 살립니다.

밝기에 대한 변화를 카툰 쇼이딩의 반대로 다음과 같이 밝은 부분은 어두운 곳이 샘플링 되고 반

대로 어두운 부분은 밝은 곳이 샘플링 되도록 만듭니다. 그리고 의도적으로 흰색 부분을 많이 넣어서 여백의 미를 만들 수 있도록 합니다.



<수목화 쉐이딩 텍스처>

쉐이더 작성은 카툰 쉐이더와 같습니다. 단지 Lambert 확산 값(Dif)과 Phong 반사 값(Spc)을 곱하고 수목화 쉐이딩 텍스처에서 샘플링 하는 것이 차이입니다.

```
float4 InkPxlInv(SvsOut In) : COLOR0
{
    float4 Out = 1;
    ...
    Nor.xyz = Nrp.xyz;
    Nor     = normalize(Nor);

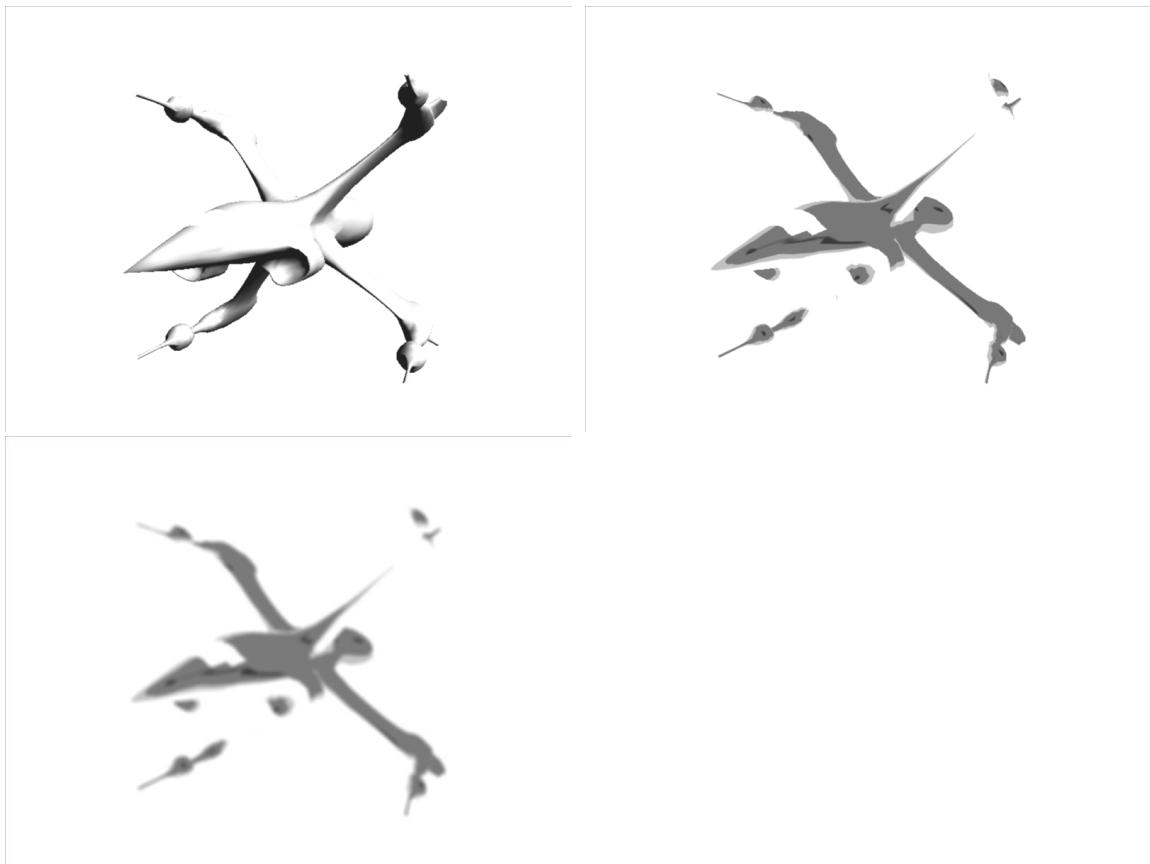
    Dif     = saturate( dot(Nor, Lgt));
    Spc     = saturate((dot(Rfc, Eye) + 1) *.5);
    Spc     = pow( Spc, m_fShrp);
    Spc     *= 7;

    // Lambert 값과 Phong 반사 값을 곱한다.
    Out = Dif * Spc;

    // 수목화 쉐이딩 텍스처에서 샘플링 한다.
    Out = tex2D(smp0, float2(Out.x, 0.5f));

    return Out;
}
```

이렇게 수목화 쉐이딩 텍스처를 처리하고 나서 흐림 효과를 적용하면 다음의 오른쪽 그림을 얻을 수 있습니다.



<[h4\\_07\\_out\\_line2\\_ink2\\_ink.zip](#): 일반 조명, 수묵화 텍스처, 흐림 효과>

이제 앞의 외곽선과 합치는 일만 남아 있습니다. 둘을 합치고 배경 텍스처를 혼합하면 다음과 같은 장면을 만들어냅니다.





<[h4\\_07\\_out\\_line2\\_ink3.zip](#)>